Computer Structures with the ARM Cortex-M0

Geoffrey Brown Bryce Himebaugh

February 12, 2016

Contents

List of Examples

List of Exercises

Preface

We assume that you are familiar with and have written a number of programs in C, and are comfortable with the following aspects of the C language

- Names and Scope
- Arrays, Strings, and Pointers
- Structures
- C types and type casting
- Loops and procedures
- Multi-file programs

Chapter 1

Introduction

This is an introductory text for the Computer Structures (C335) course taught to juniors at Indiana University. Students taking this course will have had two semester-length programming courses, a class in discrete mathematics, and a C/Unix short course. The fundamental purpose of Computer Structures is to provide a thorough understanding of how a computer executes a program and how that program, using the hardware resources of the processor, interacts with the world.

The Computer Structures course is logically divided into two parallel components – the lecture and its corresponding programming assignments; and a hands-on laboratory. The lectures examine in depth how a C program is implemented in a processor's instruction set and how that instruction set is executed by a processor. The lecture also provides a general discussion of input/output (I/O) using memory-mapped I/O, interrupts and DMA for interacting with peripheral devices. The laboratory, focuses on programming a specific embedded device, the STM32F3, to interact with the physical world.

In the laboratory, the students use memory-mapped I/O, interrupts, and DMA to control real hardware devices; and use common communication protocols such as I2C, SPI, and Serial (UART) to communicate with off-chip devices such as LCD displays, accelerometers, and SD memory cards. The laboratory is structured as a series of experiments that introduce the various communication protocols, devices that interact through those protocols, and ways of structuring code to support efficient interaction (e.g. interrupts and DMA). The culmination of the laboratory is a pair project to design and build an interactive embedded game using all of the various devices and techniques covered through the prior experiments.

In addition to the core material, the laboratory and homework are designed to teach some basic design principles. Some of this is tool oriented; for example, the use of: version control (git), hardware and software debuggers (logic analyzer, and gdb), the compiler tool-chain (linker and binary utilities), and build tools (make). We also try to teach students how to structure projects in terms of separately compiled and tested modules; and through the use of test harnesses to support development of key components on a workstation, where they are easy to test and debug, prior to deploying them on embedded hardware where they are difficult to test and debug. While the use of IDEs is common, we explicitly do not use one for this course – one of our objectives is for you to understand everything "under the hood"; while IDEs can greatly simplify the process of software development, they do this by automating and hiding many of the steps that we want you to understand.

This book begins begins with a discussion of storage – the organization of memory, how objects are laid out in memory, and how program address space is organized by the linker. We discuss the representation of information in binary form – numbers (unsigned, signed, and characters). We then dive into a basic study of a processor as an interpreter of assembly language instructions – through a series of homework assignment the students are asked to develop an interpreter for the ARM Cortex-M0 assembly language. [1] Initially, our focus is on the programmer resources – memory and registers; and their use by assembly instructions. We then systematically study C, its translation into assembly, and its use of the memory and registers. Although we do consider the encoding of assembly language into machine code, we do this rather late and with the goal of understanding the fundamental trade-off between instruction size and the ability to encode information (e.g. constants and register names) in a single instruction.

While this book will examine assembly language in depth, our goal is not to make students into expert assembly programmers – in fact very little programming is performed at the assembly level in contemporary practice. Rather, we use assembly language as a way to understand the C execution model. Unlike most books on assembly language, we examine in depth the "application binary interface" (ABI) that serves as contract between assembly programming and compiler by defining how objects are represented in memory, and how registers and the stack are used to both to interact with procedures and by procedures for temporary storage, and the responsibilities

[1] The processor used in this course, the STM32F303, has a Cortex-M4 core; however, the M0 instruction set is both simpler to understand and a strict subset of the M4 assembly language

of a procedure to save and restore state. Our rational is that the most common use-case for assembly is for small, frequently used, key routines accessed from a compiled language.

Our reason for focusing on C rather than more complex languages such as C++ is simple – we want all aspects of resource usage to be fully exposed to the students. Furthermore, C is still the most important language for embedded programming. We examine in some depth the GNU tool chain including compiler, linker, assembler, and binary utilities.

Our target architecture for this course is the STM32 family of embedded processors based upon the Cortex-M ARM cores. ARM processors, in their many forms, are the most prevalent 32-processors on the planet. A high proportion of cell phones and smart devices utilize ARM processor cores. Our focus on embedded processing allows us to ignore topics that are better postponed to Operating Systems and Computer Architecture classes; for example, virtual memory and processes. A major benefit of our focus is that, between the lecture and laboratory, this course will enable students to be serious players in developing devices for the emerging Internet of Things (IoT).

While there are simpler paths to building embedded devices (e.g. Arduino), those tend to focus on restricted programming models and using predefined software libraries. The simplicity of the Arduino program model comes from hiding exactly the details that are the focus of this course. A student completing the Computer Structures, should find Arduino environments quite easy; furthermore, they will be in a position to develop new libraries to support Arduino devices.

1.1 Lab Environment

The laboratory for Computer Structures has evolved significantly over time. For a number of years we used a dedicated robot, based upon a childrens' toy (Goofy Giggles), for our laboratory experiments. While the students enjoyed the laboratory, and we where able to meaningfully cover many of the key ideas of software/hardware interaction, it was our belief that this approach has a fatal flaw – it was not obvious how the students could translate the knowledge obtained into building novel hardware devices of their own.

Simultaneously, we observed the market being flooded with small device level "breakout" boards supporting easy prototyping with a wide variety of physical devices. Such boards are available domestically through SparkFun, AdaFruit, Pololu and others; as well as internationally through ebay. Given

this ready supply of components usable by students with little electrical engineering background, we redesigned our laboratory around a representative sampling of such devices. Our initial set of devices is illustrated in Figure 1.1 along with their approximate price. A key component is the extremely capable STM32F3 Discovery board which has an embedded hardware debugger interface as well as all the components necessary for an 9-degree inertial measurement unit (compass, accelerometer, and gyroscope).

In our laboratory, we use a carrier board (Figure 1.2) to provide a rigid platform for these devices. Such a board is unnecessary in general, but extremely useful in an environment where it is necessary to store experimental setups between use. We still require students to wire together the building blocks, but provide a stable platform for their use.

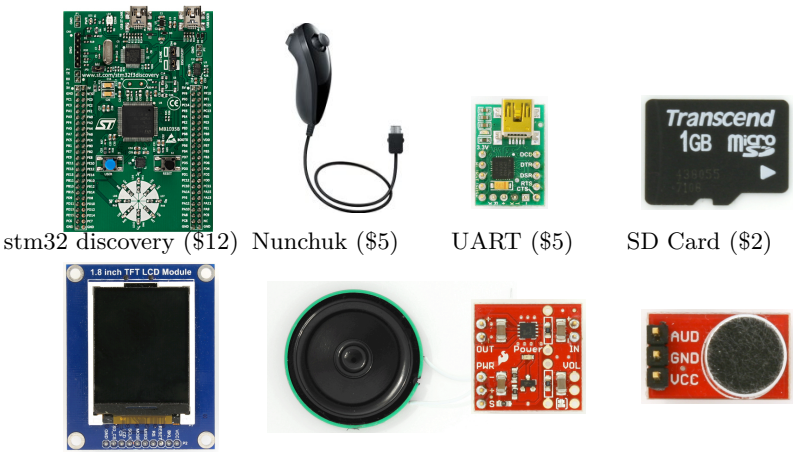


Figure 1.1: Laboratory Building Blocks

Because the processor board we use has a built-in hardware debugger interface, very little laboratory equipment is necessary beyond a Linux workstation. We do teach our students how to use the inexpensive logic analyzers developed by Saleae; for the low-speed hardware protocols we utilize (SPI, I2C, serial), these logic analyzers are more than adequate. Furthermore, the software interface is both easy to use, and powerful – the built-in protocol analyzers are wonderful. The "classic" logic analyzer and a trace illustrating one I2C transaction are illustrated in Figure 1.3.

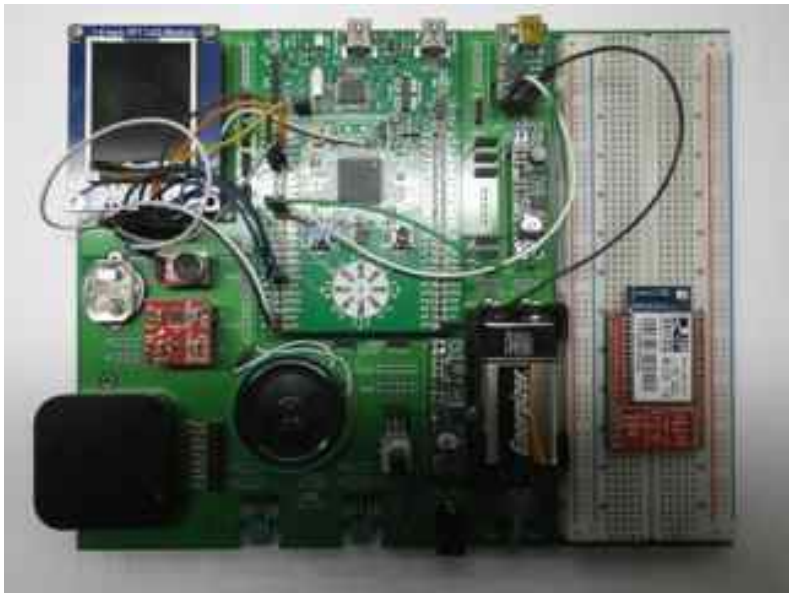


Figure 1.2: Laboratory Building Blocks

Figure 1.3: Saleae Logic Analyzer and I2C Trace

1.2 Software Hierarchy

This book is primarily concerned with software/hardware hierarchy illustrated in Figure 1.4. At the bottom of the stack is the hardware; i.e. the processor and its associated input/output (I/O) devices. The remaining layers that we consider in this course are machine code, assembly language, and C. Many high-level languages (for example python) are implemented in part using C programs.

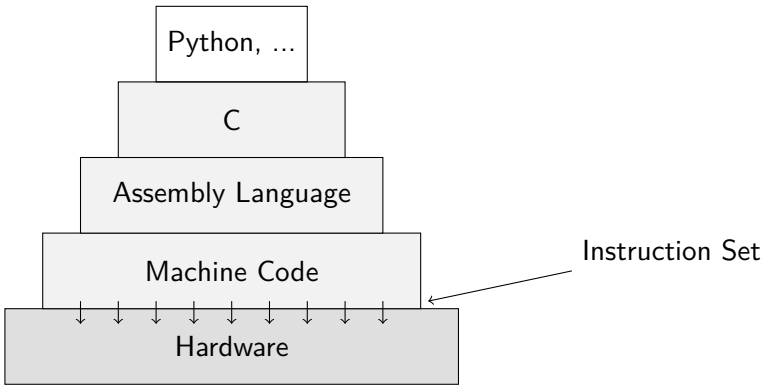


Figure 1.4: Software/Hardware Hierarchy

The processor is controlled by binary programs written in its instruction set – we'll have a great deal more to say about this later. The binary encoding of the instruction set is designed for efficiency of execution rather than ease of programming. In general, the first programming level in which humans are engaged is assembly language. At its simplest, assembly language is a textual form of the underlying machine language; however, most assembly languages provide features to simplify the task of programming. For example, assembly languages allow programmers to express memory locations symbolically – using labels – rather than having to calculate the physical address. The final level that we will be concerned with in this course is the C program. While assembly language is by definition machine specific, C is a portable language.

C was initially designed by Dennis Ritchie at AT&T Bell laboratories for the development of the Unix Operating System with the goal of replacing assembly language, which is machine specific, by a more portable language that could be efficiently translated into the assembly language of various comput-

ers. [2] Although the C language has evolved over the years, a strong connection remains between a C program and its use of the underlying machine resources. For application programs this can be a liability – the control afforded by the C model comes at the expense of greater opportunity for bugs as well as the significant effort required to manage memory resources. For the embedded world, where resources are both tight and expensive, this fine-grained control is essential.

```
int counter;

int counterInc(void){
  return counter++;
}
```

```
counterInc:
0 024B     ldr  r3, .L2       @  r3 = &counter
2 1868     ldr  r0, [r3]      @  r0 = *((int *) r3)
4 421C     adds r2, r0, 1     @  r2 = r0 + 1
6 1A60     str  r2, [r3]      @ *((int *) r3) = r2
8 7047     bx lr              @ return, value in r0
a C046     .align  2          @ choose next 4 byte address
     .L2:
c 00000000 .word   counter    @ &counter will be stored here
```

Figure 1.5: C/Assembly Example

In order to get a sense of this hierarchy, consider the examples in Figure 1.5. This figure contains two major parts. The top box contains a simple C procedure that increments a global variable `counter`. The lower box contains, from left to right, memory addresses (0,4,8,..), machine instructions in hexadecimal notation (0x4B02, 0x1C42)[3] assembly code (the textual representation of the machine instructions), and finally comments relating the machine code to the original C. The assembly program loads the address of `counter` in the first instruction – reading the saved global address from the location at label .L2 (the actual address will be placed here by the linker). The next three instructions read the value of counter by dereferencing the address, update this value, and write the new value back to the address of

[2] C.A.R (Tony) Hoare famously quipped that C would be a fine language if they removed all the PDP-11 assembly instructions.

[3] The assembly listing is in byte order; 0x4B02 is the instruction consisting of bytes 02,4B.

counter (again by pointer dereferencing). Finally, the last instruction returns the old value of `counter` (as is expected with the post increment). [4]

While it would be unreasonable to expect that you fully understand this example at this stage, it is illustrative both of the multiple levels at which this course operates, and of the connections between C, assembly, and instructions that we expect you will build through this course.

An important differentiator of Computer Structures is the emphasis that we place upon understanding the execution model of C programs. One example of that is the (simplified) memory model of an executing C program illustrated in Figure 1.6. This model illustrates four major memory regions – stack, heap, data, and code. The stack is used during execution to allocate temporary storage – every invocation of a procedure may allocate space on the stack. For example, variables declared within a procedure body are usually allocated space on the stack. The data region consists of static variables declared by a program – e.g. those declared outside of a procedure. The space for data is allocated at compile time by the program linker. The code area consists of the machine instructions of a program. Finally, the heap consists of dynamic storage allocated at runtime by allocators such as `malloc`. This model is simplified in (at least) two ways – it ignores other runtime regions such as shared libraries, and the actual relative position of the regions in physical memory may be quite different. Nevertheless, we will return to this abstract model frequently to discuss resources.

Compiler/Assembler/Linker – need a diagram and discussion of the various files here.

Throughout the lectures, we relate this model to the text of the a C program, to the sections of the object file, to the assembly code, and to executing programs. We discuss as some length the roles of both the linker and runtime in allocating memory and where information "lives" in an executing program.

[4] post-increment and pre-decrement are examples of the PDP-11 instructions to which Hoare referred

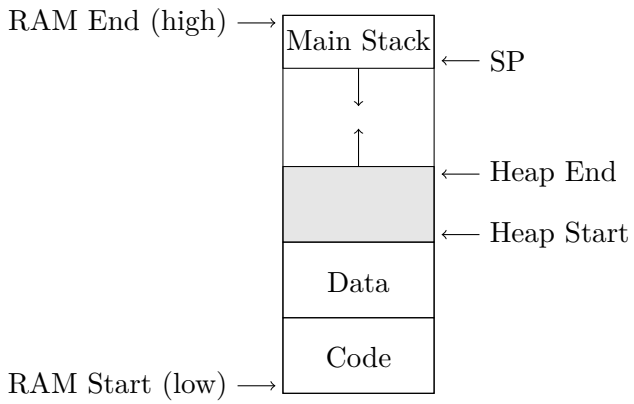


Figure 1.6: C Memory Model (Simplified)

1.3 Cortex-M Processors

In both lectures and laboratory we examine ARM Cortex-M based processor cores. There are (at least) three distinct Cortex-M cores – the M0, M3, and M4. these form an upward compatible family of instruction sets with M0 being the simplest and M4 the most complex. By upward compatible, we mean that programs compiled for M0 cores, will execute upon M3 and M4 cores and programs compiled for M3 cores will executed on M4 cores. The M0 instruction set forms a complete language and is the focus of this book. The M3 and M4 cores add support for additional data operations as well as numerous 32-bit instructions (the M0 is almost exclusively a 16-bit instruction set).

The Cortex-M processors are used in a extremely wide variety of embedded processors from several manufacturers. These processors differ in the set of hardware peripherals and memories they provide. Binary compatibility breaks down at the core boundary – with the exception of a tiny set of "standard" peripherals included in the Cortex-M core, all the other peripherals are specific to individual manufacturers and chip families. In the laboratory, we use the STM32f303 processor with is part of the STM32f3 family of components from ST Microelectronics. The STM32f3 processors contains Cortex-M4 cores, which, as we have pointed out, provide a superset of the the Cortex-M0 instructions.

A "generic" Cortex-M based processor is illustrated in Figure 1.7. As

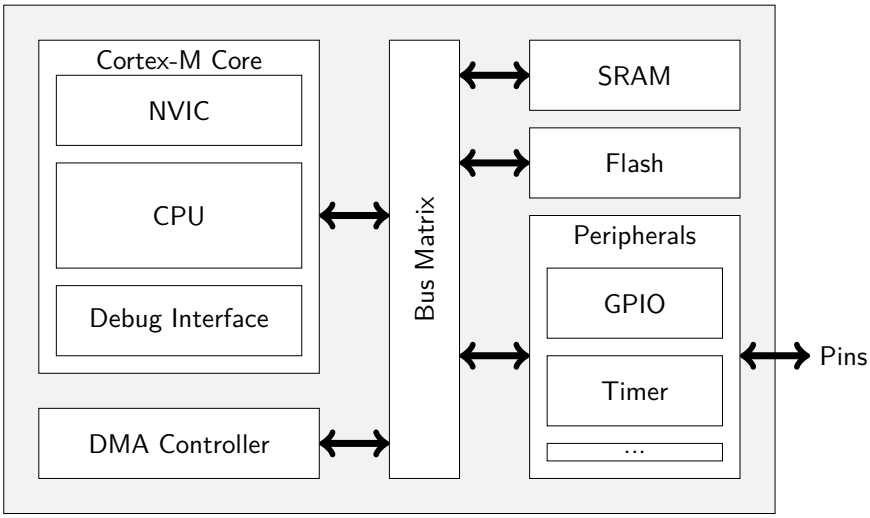


Figure 1.7: Generic Cortex-M System

illustrated, this consists of a Cortex-M core, memory (static ram and flash), a set of device specific peripherals, and a DMA controller. The core includes the CPU (central processing unit), which executes machine instructions, an interrupt controller (NVIC), and a debugger interface; these and other components are part of every Cortex-M core.

The Cortex-M core communicates with memory and peripherals through a *bus matrix*, which enables data to be routed across a fixed set of connections; in some cases multiple simultaneous transfers can occur. A key idea is that, from the perspective of the CPU, peripherals and memory all look the same – they all accessed by a program by reading and writing locations that are addressed by pointers. This approach to input and output is called *memory-mapped I/O*.

The Cortex-M processors all have 32-bit address spaces, which means that addresses (pointer values) fall in the range $0..(2^{32}\ldots 1)$ or, in hexadecimal 0x00000000 …0xFFFFFFFF. As noted above, this address space is shared with both memory and peripherals. The general address map defined for the Cortex-M0 processors and others is illustrated in Figure 1.8. In our generic model, the SRAM (random access memory) would be access as a subset of the address range 0x20000000…0x3FFFFFFF and FLASH (read only memory) in the range 0x0000000…0x1FFFFFFF.

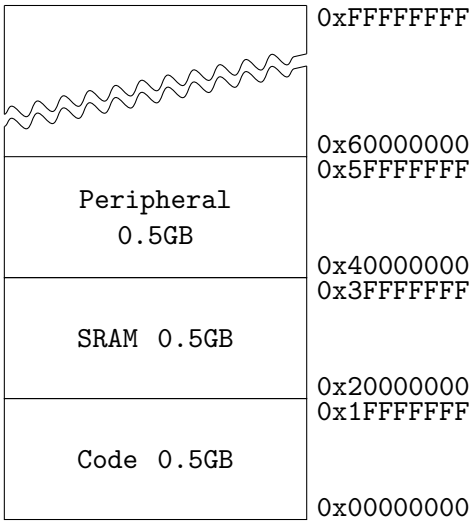


Figure 1.8: Cortex-M Address Map

Figure 1.7 includes a few typical peripherals – GPIO or general purpose I/O which is typically used to read values from or write values to pins on the chip boundaries, and a timer which can be used to synchronize I/O events with time and, by a program, to measure time. These devices are controlled by reading and writing chip specific addresses in the range 0x40000000–0x5FFFFFFF.

Finally, our generic example includes a DMA (direct memory access) controller. DMA controllers can be programmed to transfer data between memory and peripherals synchronously with either external or temporal events. DMA controllers are themselves specialized peripherals that are under program control and are accessed by reading from and writing to addresses in the peripheral address space.

1.4 A Cortex-M Based System

As a concrete example of a Cortex-M based core, consider the STM32F303 that we use in laboratory. This is a very capable chip that includes a Cortex-M4 core as well as a large variety of I/O devices that communicate with the core over a bus matrix. A fragment of this system is illustrated in Figure 1.9. This fragment includes the DMA controllers, Analog/Digital Convert-

ers (ADC), memories, general purpose I/O (GPIO), timers, SPI, and USART devices. As this fragment illustrates, this is a complicated chip and in this course we only touch the surface. Key ideas that are discussed are the configuration and control of devices, configuration of DMA, and interrupts.

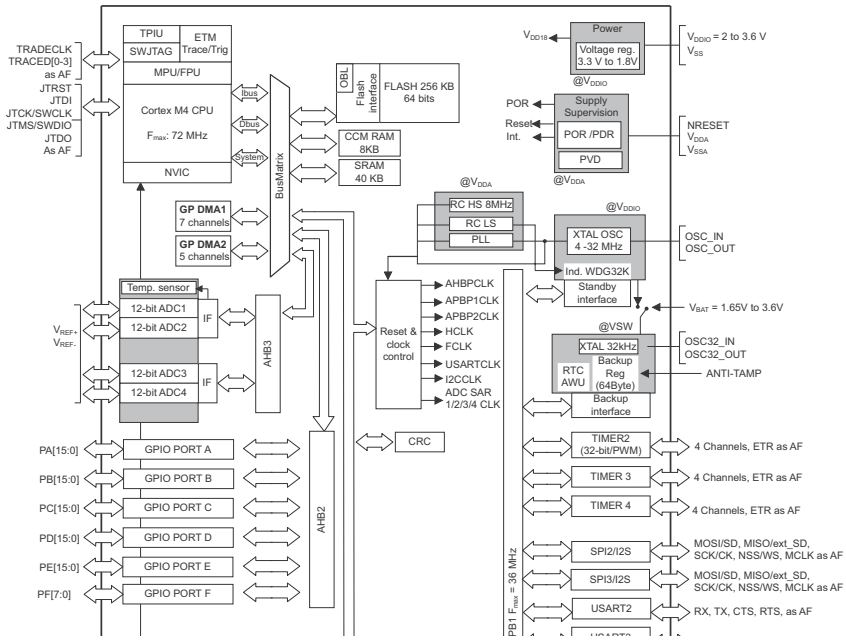


Figure 1.9: STM32F303 Block Diagram

Perhaps a more useful abstraction of the STM32F303 system is illustrated in Figure 1.10. This figure illustrates the physical connections between the various components. For example, the processor core uses the I-bus to fetch instructions from Flash, SRAM, or CCM Ram. Simultaneously, it may use the D-bus to interact with devices such as the GPIO or ADC.

As mentioned previously, a key idea that is discussed in both lecture and lab is memory-mapped I/O. Consider again Figure 1.8 which illustrates the static partition of the 4GB address space of the Cortex-M processors. A fragment of the STM32F303 peripheral address space is illustrated in Figure 1.11. [5]

This fragment shows where the ADC1,ADC2, GPIOA, USART1, and SPI1 devices are located (address range) and the interconnection path used

[5]STM Doc ID 023353 Rev11

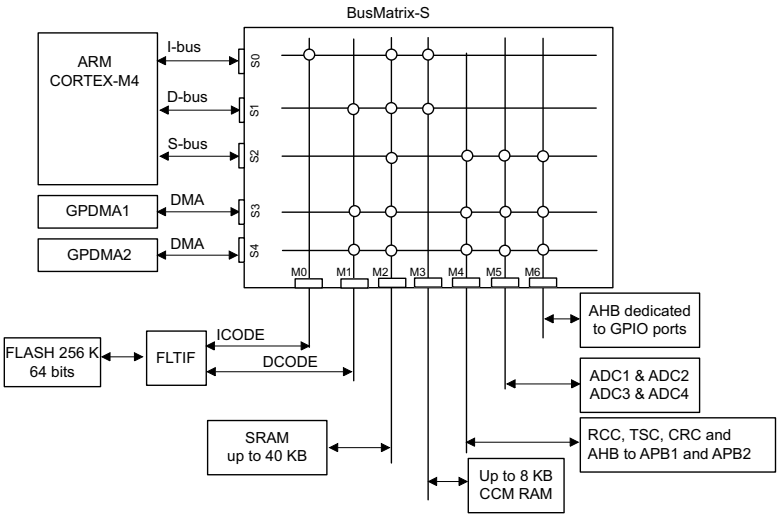


Figure 1.10: STM32F303xC System Architecture

to access them (Bus). The ADC devices are analog to digital converters, that enable the processor to measure an external voltage; the USART is a serial device used to communicate with other hardware devices; and the SPI device is a hardware interface we use in lab. Knowing which interconnection path is used is important for understanding performance limits – for example, GPIOA and ADC1 are on separate buses. It would be feasible to use DMA to transfer samples from the A/D converter to the pins on GPIOA.

Bus	Address Range	Peripheral
AHB3	0x5000 0000 - 0x5000 03FF	ADC1-ADC2
AHB2	0x4800 0000 - 0x4800 03FF	GPIOA
APB2	0x4001 3800 - 0x4001 3BFF	USART1
APB2	0x4001 3000 - 0x4001 33FF	SPI1

Figure 1.11: Example STM32F303 Peripheral Addresses

Every I/O device is comprised of a number of separate memory-mapped registers, which we can access through a C program. Figure 1.12 illustrates a C structure defining a subset of the GPIO registers, and a code fragment accessing this structure. Two registers are shown – one for reading the GPIO

pins (IDR), and one for writing the pins (ODR). The code fragment casts the address of GPIOA (Figure 1.11 to a pointer of type `GPIO_TypeDev*`. The address of each register is relative to the start of this structure. Finally, the procedure `GPIO_ReadInputData` reads the pins of the GPIO device passed as a parameter.

```
typedef struct {
...
__IO uint16_t IDR;  // GPIO port input data register
...
__IO uint16_t ODR;  // GPIO port output data register
...
} GPIO_TypeDef;
```

```
#define GPIOA_BASE  0x48000000
#define GPIOA ((GPIO_TypeDev *) GPIOA_BASE

uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx) {
  return GPIOx->IDR;
}
```

Figure 1.12: Example STM32F303 Peripheral Addresses

1.5 Required Tools

- gnu arm toolchain
- qemu-arm
- git

The examples and exercises in this book require access to several open-source tools. These include the compiler/assembler/linker/debugger toolchain – the gnu-arm tools, the QEMU system simulator, and the git revision control system. In this section we explain how to download and install these tools on Linux and os X systems. We do not advise that you attempt to install these on Windows based systems.

Chapter 2

Storage

As mentioned, C, has a relatively limited set of basic data types including various sizes of integers and floating point numbers, and pointers (addresses); and a small set of constructors for building aggregate data types including unions, structures, and arrays (strings are a special case of arrays). This is in contrast to higher-level languages that include built-in data types such as lists and dictionaries and rich type systems that admit the creation of rich data types with associated operators.

One of the most interesting characteristics of modern processors is that there are essentially no "types." Data are stored in memory and, depending upon the context, are interpreted as instructions, integers, floating point numbers, characters, etc. While the primitive operations provided by the processor may be defined assuming they operate on data of a particular type (e.g. integers), the processor has no mechanism for checking – indeed there is nothing in the data to denote type. [1]

Memory is organized as a collection of words that are accessed by reading and writing using addresses – the primitive memory operation is a pointer de-reference. As we shall see, even this is context dependent because a particular region of memory may "mapped" to a hardware device and writing a specific word might have the effect of turning on a light or motor.

[1] Systems with virtual memory system may provide permission bits to ensure that data aren't executed, but this isn't really part of the core processor.

2.1 Bits, Bytes, Words

In this section we consider the fundamental information containers provided by a processor. As you are probably aware, the fundamental building block for digital systems is the bit, or "binary digit", which can have two possible values – 0 or 1. Larger information containers are built from bits including bytes, which are groups of 8 bits, words, which are a machine dependent number of bits, and memories which are arrays of bytes or words.

Unlike bit or byte, word does not have a fixed meaning. In general a computer's *word size* is the largest number of bits that can be transferred in a basic memory access or, perhaps more precisely, the size of a memory address (or pointer). Although historically, there have been machines built with many word sizes, must modern machines have word sizes which are 2^n bytes (n = 0, 1, 2, and 4 are all common). [2] Regardless of word size N, we number bits from 0 to N-1; it is convenient to refer to bit 0 as the *least significant bit* (lsb) and bit N-1 as the *most significant bit* (msb) – these names will take on greater meaning in the context of integer encodings. This is illustrated for 32-bit and 8-bit words in Figure 2.1.

Figure 2.1: Word Bit Numbering

Consider the 8-bit quantity `1011 0100`. The msb (bit 7) is 1, and the lsb (bit 8) is 0. We will frequently need to refer to data such as this in binary form; however, long binary strings are a challenge both to remember and read unambiguously. It is common practice to use a compressed form, called hexadecimal, in which groups of 4-bits are represented by one of the 16 digits 0..9,A,B,C,D,E,F. This form is particularly convenient because conversion between hexadecimal and binary is a simple textual substitution – each hexadecimal digit is exactly 4 binary digits.

[2] The DEC pdp-8 had a 12-bit word, and the DEC-10 had a 36 bit word. The latter is significant because 36 bits is sufficient to represent integers to 10 decimal places.

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Thus

$$(126)_{10} \;=\; (0111\ 1110)_2 \;=\; (7E)_{16}$$

The main reason we use hexadecimal is to make it easier to type, transcribe, and remember large binary numbers – it's far easier to remember a string of 8 hexadecimal digits than 32 binary ones. It is extremely important that you memorize the table defining hexadecimal digits – conversion between binary and hexadecimal is something you will do frequently.

2.2 Word Size

The definition of a "word" is machine dependent – many different sizes have been used over the history of computing; although 8, 16, 32, and 64 are the primary ones in use today. The two most important characteristics of a machine's word size are the size of the memory that can be addressed, and the size of the basic integer or logical operations that can be performed in a single machine instruction.

The word size matters – for a word of size `n`, the word size determines:

- The memory address range: $0..2^n - 1$.
- Primitive integer range:
  - signed integers: $-2^{n-1}..2^{n-1} - 1$
  - unsigned integers: $0..2^n - 1$.
  - Operations on larger sizes require multiple instructions

In general, one must be careful about moving word-size dependent code from one platform to another. The C standard include file `<limits.h>` provides easy access to platform specific limits; however, it has become common practice to utilize types such as `int32_t` and `uint16_t` to explicitly define word size assumptions in code where that matters. One must also be extremely careful when casting values between word sizes to ensure that the old value can be represented faithfully in the new type. While such C issues are beyond the scope of this course, the material covered in this book should provide a solid foundation for understanding how to recognize and repair size dependencies in C programs.

To get a sense of the differences in C type representation for various processors, consider Figure 2.2.

C Data Type	IA64	ARM 32bit	Required
char	1	1	minimum size to hold a character
short	2	2	at least 2 bytes
int	4	4	at least 4 bytes
long int	8	4	at least 4 bytes
long long	8	8	at least 8 bytes
float	4	4	commonly 4 bytes (IEEE)
double	8	8	commonly 8 bytes (IEEE)
void *	8	4	machine dependent

Figure 2.2: C Data Type Sizes (in bytes) for Various Architectures

2.3 Byte Addressable Memory

Most processors now define memory as an array of bytes. Thus, pointer (addresses), refer to a byte offset within memory. Larger sized objects such as shorts, ints, floating point numbers, etc. are stored in multiple byte address

locations. This is illustrated in Figure 2.3 where a memory with 8 locations is shown with "overlays" for 16-bit, 32-bit, and 64-bit types. Notice that in this figure the larger objects are *aligned* on natural boundaries. For example, 16-bit objects are at even addresses, and 64-bit objects at addresses divisible by 8 (0, 8, 16,...). *Unaligned* objects are those that do not fall on natural address boundaries – many processors will either refuse to implement unaligned memory accesses or will silently fail to correctly implement the memory reference. A few processors, such as the x86 family, implement unaligned memory accesses, but at a significant performance penalty. The reason for this is that, internally, processors read and write memory in blocks that are a multiple word size. Accesses that cross such blocks require multiple reads/writes for a single operation, which can cause significant problems in systems supporting virtual memory because the first access might succeed while the second causes a page fault forcing the instruction to be restarted mid-execution.

The ARM Cortex-M0, which is the primary focus for this book, requires that half-word objects be aligned on half-word boundaries (even byte addresses) and that word or multiple word addresses be aligned on word boundaries (0,4,8, etc.). Some ARM variants provide some support unaligned memory accesses; the Cortex-M0 will raise a `HardFault` exception on any attempt to perform an unaligned memory access. ARM has pushed for a move to a "unified" assembly model in which code written for one processor is binary compatible with code written for more capable processors. Some of the ARM processors support double-word memory accesses and therefore require that 64-bit objects be aligned on 8-byte boundaries. In order to preserve compatibility, we will adhere to this alignment restriction even though the Cortex-M0 does not require it.

Byte Order

Byte addressable memory introduces one source of frequent confusion – how to address (number) the bytes within a word. Unfortunately, there are two conventions – big-endian, in which the left-most (most significant) byte appears at address offset 0; and little endian, in which the right-most (least significant) byte appears at address offset 0. Little-endian byte order (Figure 2.4) is notably used by the x86 processor family; big-endian byte order (Figure 2.5) is used by MIPS processors, and, the internet protocols. The ARM processors can support either order, but we use them in little-endian configuration. The acronym `msb` stands for "most significant bit" (MSB stands for most significant byte), and the acronym `lsb` stands for "least significant

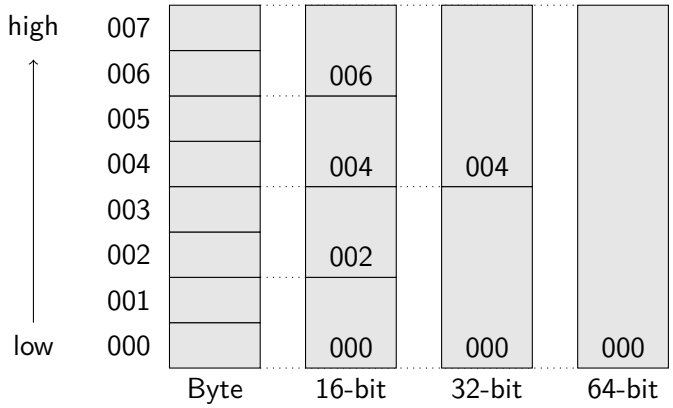


Figure 2.3: Byte Addressable Memory

bit." In the big-endian world, bit 7 of byte 0 is the msb while in the little-endian world, bit 0 of byte 0 is the lsb.

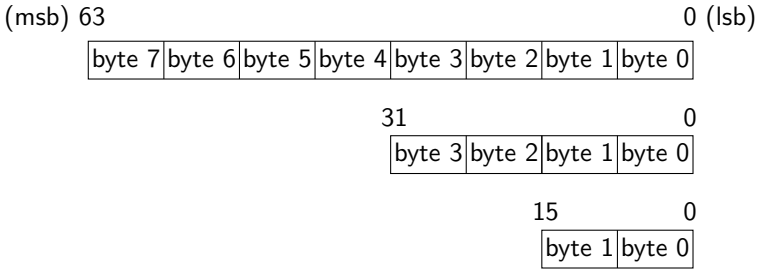


Figure 2.4: Little Endian Byte Order

Students are frequently confused about byte order – again byte order refers to the bytes within a word and not the bits within a byte. When moving data between systems, it is frequently necessary to swap byte order. Consider the example in Figure 2.6. The same 32-bit word `0xAB987654` is shown in both big and little-endian form. [3] Notice that the address of the low-order byte starts on a multiple of 4. To move between the two formats, it is necessary to swap bytes 0 and 3 and swap bytes 1 and 2. For a 16-bit word, one needs

[3] We will review the hexadecimal format 0xAB... in a subsequent chapter.

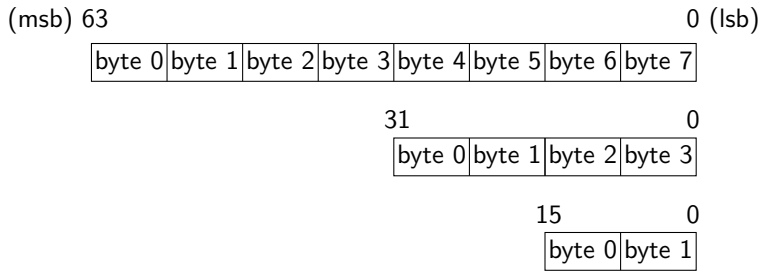


Figure 2.5: Big Endian Byte Order

to swap bytes 0 and 1. For a 64-bit word, one needs to swap bytes 0 and 7, 1 and 6, 2 and 5, 3 and 4.

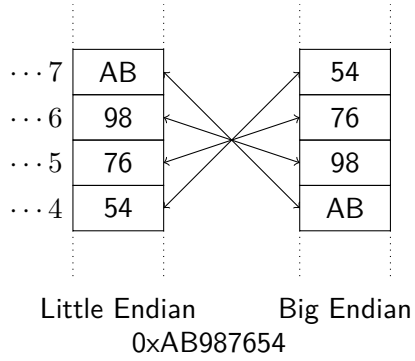


Figure 2.6: Byte Swapping

Byte swapping is such a frequent occurrence, that the Cortex-M processors include special instructions for swapping bytes in words and half-words (16-bits). One problem with writing portable code is that the same code should run correctly irrespective of the byte-order of the underlying architecture. The standard C include file `<machine/endian.h>` includes macros to define the byte order of the target machine.

2.4 Arrays and Pointers

In high-level programming languages such as Java, the physical manifestation of an object is hidden from the programmer – the program creates

Example 2.1: Pointers and Arrays

```
int v;
void foo(void) {
   unsigned char *p = (unsigned char *) &v;
   p[0] = 0x10;
   p[1] = 0x20;
   p[2] = 0x30;
   p[3] = 0x40;
}
```

and uses objects and the garbage collector reclaims them once the program no longer needs (references) them. While objects exist as blocks of memory, the programmer has no mechanism to directly reference this memory – the only access mechanism is through the methods of the object.

In contrast, C exposes the underlying representation to world. Consider C program fragment in Example 2.1. The program declares a variable `v` of type `int`. In a language such as Java, the only actions we could take with this object are to read integer values from it and write integer values to it. In this C fragment, the procedure `foo`, creates a pointer to `v` – literally the address of the memory implementing `v`, and casts that to a pointer of type `unsigned char`. The procedure then writes values to each of the four bytes comprising `v`. Thus C, unlike Java, allows the programmer to manipulate the representation of data.

Java does have the ability to pass *references* to objects, but references do not provide additional powers to manipulate the objects to which they refer – these are still restricted to the methods defined for those objects. Hence Java references are fundamentally different that pointers. In fact, the memory used to implement a Java object may move dynamically during execution due to garbage collection, yet a reference to the object remains valid. This, in itself, should be an indication that a Java reference is distinctly different than a C pointer.

Returning to the example above, notice that once `p` is assigned a value (the address of `v`), the program treats `p` as an array and uses array syntax to address the individual bytes of memory. If we were to print the contents of v – as in

```
printf("0x%x", v);
```

The result would be either

```
0x40302010
```

or The result would be either

```
0x10203040
```

Depending upon the native byte-order.

We could equally well have used pointer arithmetic to perform the same action.

```
*p     = 0x10;
*(p+1) = 0x20;
*(p+2) = 0x30;
*(p+3) = 0x40;
```

This approach can lead to some confusion. Suppose that we wish to access an integer array using pointer arithmetic, then the following two approaches are equivalent

```
int *q =  ...

q[0]   = 1;
q[1]   = 2;
q[2]   = 3;
```

and

```
*q     = 1;
*(q+1) = 2;
*(q+2) = 3;
```

The advantage to the array syntax is that it handles the address calculation in a natural manner, whereas, with pointer arithmetic we have to be cognizant of the size of the objects we are addressing. A further difference is that array names provide a mechanism for static memory allocation in C. For example,

```
int a[7];
```

allocates a block of memory large enough to hold seven integers. The name of this block of elements is `a` and its elements are `a[0]` ... `a[6]`. We can reference the address of this block of memory as

```
int *p = a;
```

or

```
int *p = &a[0];
```

Thus pointers and arrays are not identical – an array can *decay* to a pointer by assigning the array name to a pointer. It is essential that you be completely comfortable with both arrays and pointers before tackling assembly language programming. At the assembly language level all objects are accessed via pointers – variable names at the assembly level are just labels for memory locations.

2.5 Structures and Unions

Structures and Unions frequently cause confusion with students. Here is the important distinction – all of the fields of a structure occupy different areas of memory within the structure while all of the fields of a union occupy the same memory with overlapping locations. Thus, modifying a structure field has no impact on the other fields of the structure, while modifying a union field potentially affects *all* the other union fields.

In this section we consider the relationship of pointers to a structure (union) and the fields of a structure (union). Consider Figure 2.7. Assuming four-byte integers, two-byte shorts, and eight-byte long long, the layout in memory for this structure is illustrated in both big-endian and little-endian form. The vertical axis is labeled with the word offset and the horizontal axis with the byte offset. For byte-addressed memory, the address of a a byte in this figure is computed by multiplying the word offset by four and adding the byte offset. For example, in the little endian figure, byte `b[1]`, which is the most significant byte of `b`, is at offset 5 from the beginning of the structure. Notice that the starting address of a field, is independent of whether the structure is in little or big-endian form; however, the bytes order within a field differs with endianess.

In a structure, the offset of a field from the structure start is computed

Exercise 2.1: Structure Layout

Draw the layout in big and little-endian form for the following structure:

```
struct {
       char a;
       int  b;
       char c;
       double d;
}
```

```
typedef struct {
        int a;
        short b;
        long long d;
        } exS;
exS x;
```

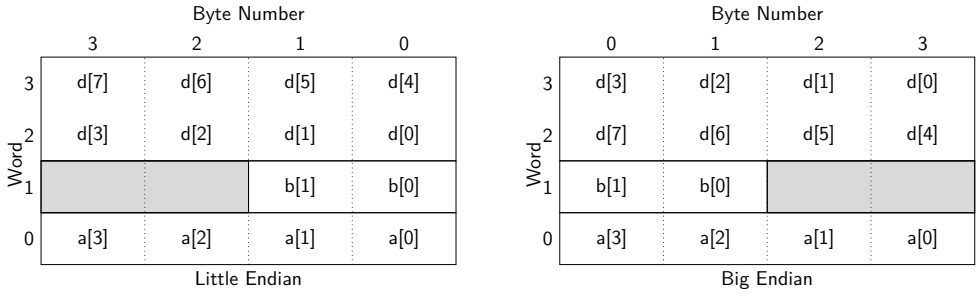


Figure 2.7: Structure Layout

as follows.

- Add the offset and size of the preceding field
- Round up to the nearest address that satisfies the alignment requirements of the field (e.g. a multiple of 8 for a `long long`).

The overall size of a structure is a multiple of the maximum alignment requirements of its fields. To understand the seemingly wasteful requirement, consider the following:

```
struct {
          long long ll;
          int i;
        } a[2];
```

The alignment requirements for first array element could be met with a 12-byte size, but then the second element would be incorrectly aligned.

The C libraries provide two macros that may be used to determine the offset and size of a field – `offsetof` and `sizeof`. This last rule ensures that, for an array of structures, each every structure and every field satisfies its alignment requirement.

Consider the following C union.

```
union {
       int a;
       char b;
       long long c;
      }
```

All three fields have the same offset from the beginning of the union – 0. The size of the union is the size of its largest field. While the Cortex-M0 will execute properly if this union is aligned on 4-byte boundaries, the current ARM ABI requires 8-byte boundaries to ensure interoperability with other members of the ARM family.

2.6 Bitfields

The C language includes *bitfield* operations that allow the manipulation of binary data at the level of individual bits. These operations include:

and `A & B` returns the bitwise and of operands `A` and `B`.

or `A | B` returns the bitwise or of operands `A` and `B`.

xor `A B` returns the bitwise exclusive or of operands `A` and `B`.

not `A` returns the bitwise inverse of operand `A`.

The operations are particularly useful for reading and writing individual bits from a packed data format (for example IP protocol packets). For example, to test bit 7 of a word, one can execute:

```
            if (i & 0x80) ...
or
            if (i & (1<<7)) ...
```

The first form uses a hexadecimal constant and the second a constant computed by the compiler. Example 2.2 illustrates the exclusive-or operation. The other bitwise operators are similar.

C also includes *right shift* (`>>`) and and left shift (`<<`) operations. Example 2.3,2.4, and 2.5 illustrate the two types of shift operations. Note that

Example 2.2: C Xor

Example 2.3: C Left Shift

Example 2.4: C Right Shift (unsigned)

Example 2.5: C Right Shift (signed)

the behavior of the right shift operation depends upon whether the operands are signed or unsigned. Later we will see that this is consistent with the interpretation of right shift as division by a power of two.

All of the C bitwise operations may be applied to any signed or unsigned integer data type (e.g. char, short, int, long long).

2.7 Linker Sections

The *memory* map of a program is determined by the linker. The compiler and assembler generate object files with data and code divided among various sections. These sections include `code`, `data`, and `bss`. Both the `data` and `bss` sections contain program data, but `bss` contains that data that will be initialized to 0 at program start.

At the level of a single unlinked object files, all of these sections begin at address 0; the compiler and assembler allocate space within these abstract sections, but do not determine what physical memory will ultimately be allocated to them.

The linker combines the object files of a program together with any startup code and libraries to build a single executable file. Each of the sections (e.g. code and data) are created by concatenating the individual sections from the various binary objects into a single contiguous section – all of the data sections are combined into a single data section, all of the code sections are combined into a single code section„ etc. The linker assigns a starting address to each section, computes the addresses of all symbols relative to this starting address, and appropriately patches any symbol references in the object files.

The linker is controlled by a target-specific script. For the generic Cortex-M0, the code section is assigned to the area beginning with address 0,

the data section and bss sections are assigned to the memory area beginning with address 0x20000000.

Later, we'll see that the first portion of the code section is devoted to interrupt and exception vectors. As part of the processor boot procedure, the initial stack pointer is loaded from this area. The address for this stack pointer is defined by the linker script.

We have frequently referred to the three sections `code`, `data` and `bss`. Both the compiler and linker are free to define other sections, for example, the static constructors and destructors for C++ objects are assigned to special sections so that they can easily be found for execution at program start and exit time. When new sections are added, the linker script will need to be modified to define the appropriate memory allocation. This topic is beyond the scope of this book. We will limit our discussion to the three key sections.

Chapter 3

Memory Mapped Input/Output

The central theme of this book is how a program, written in C, is executed by a processor. Certainly, all interesting programs interact with the world external to the processor using *input/output* (I/O) devices such as displays, keyboards, and communication networks. Somewhat surprisingly, the interface between a program and these I/O devices is identical to that of memory; in particular, the individual I/O devices respond to memory read and write commands. We refer to such devices as *memory-mapped* because their interfaces are part of the memory address space of a processor. The fundamental difference between memory and I/O devices is what happens when they are read or written by a program. Memory is simply storage – reading a memory address returns the last value written to that address. In contrast, I/O has side effects; for example reading from the input *register* of a serial port may return the last value received at a serial port and writing to its output register may have the effect of transmitting a character to another computer connected to the other end of the serial port. In this chapter, we discuss memory-mapped I/O.

Registers of I/O devices are defined memory locations that, when accessed, affect the behavior of that device.

We begin with a discussion of memory and its interfaces – clearly if I/O devices share an interface with memory, then before discussing the devices, we need to be clear about that interface. While it is tempting to drill down to the bedrock of the transistors, capacitors, and other primitive devices from which all computer hardware is built, this would be an extended and unnecessary diversion from the central topic of this book. Instead we focus on a few high level concepts – combinational logic in the form of arbitrary functions over N-bit words, and latches capable of storing N-bit words.

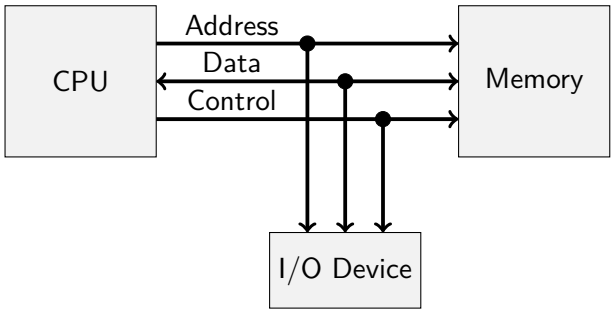


Figure 3.1: Memory Mapped I/O Model

The basic model we will develop is illustrated in Figure 3.1. This figure consists of three components – a processor (CPU), a memory, and a generic I/O device. Communication between these devices is through three sets of signals: N-bits of address, which the CPU defines when it wishes to communicate, M-bits of data which one of the CPU, Memory, or I/O device define, and some number of control lines, defined by the CPU when it wishes to communicate. In the model we develop in this chapter, the CPU may initiate two types of transfers – read and write – with either the Memory or the I/O device as specified by the address.

At the machine instruction level, this interface is completely hidden. The Cortex-M0 (and most processors) has only two basic memory access instructions. Load, which reads from "memory" address (pointer)

```
ldr r1, [r2] % r1 = *r2
```

and Store, which writes to a "memory" address:

```
str r1, [r2] % *r2 = r1
```

That's it ! There are variations of these instructions to simplify pointer calculations (e.g. accessing fields within a structure) and for sub-word access, but these are the two fundamental operations. Consider Figure 3.2 repeated from Chapter 1. The Cortex-M address space is divided into (at least) four regions. Memory load and store operations referencing addresses in the range 0x00000000-0x1FFFFFFF access the code region of memory – in the Cortex-M processors this is typically implemented as Flash storage which can be read from, but not written to by normal memory operations. Flash memory is persistent – the values are retained even when power is removed. Memory load and store operations referencing addresses in the range 0x20000000-

0x3FFFFFFF access SRAM (static random access memory) which can be both read and written, but does not persist over power cycles. The third address region 0x40000000-0x5FFFFFFF is reserved for manufacturer specific peripherals – the subject of this chapter. The final, top portion of the address space is used for some ARM defined peripherals such as the system timer and interrupt controller.

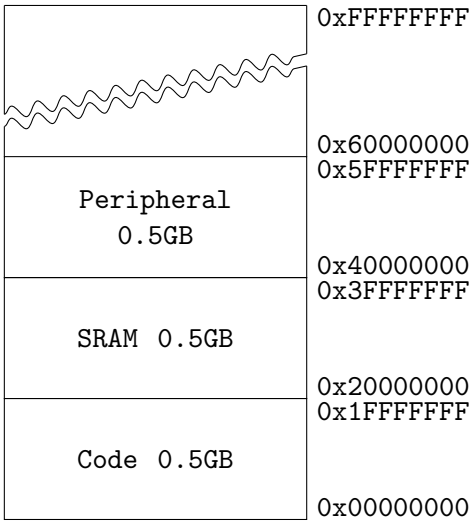


Figure 3.2: Cortex-M Address Map

As discussed in Chapter 1, in the laboratory associated with Computer Structures at Indiana University, we use the STM32F303 processor which has a Cortex-M4 core as well as a large number of ST Microelectronics proprietary peripherals. In Figure 1.11 we showed how some of these peripherals are assigned specific portions of the peripheral address space. We went on to discuss how the interface to one of these (general purpose I/O) can be defined as a C structure and the device registers accessed by reading and writing structure fields.

In this chapter, we will discuss two types of peripherals – general purpose I/O and serial, although in a somewhat generic context rather than the ST Microelectronics specific devices. While there is widespread commonality in the *types* of peripherals provided by various processors, the implementation specific details differ widely. You have probably experienced this phenomenon when adding a new peripheral, for example a WiFi interface, to your computer

and found that the operating system requires a manufacturer specific driver. The driver provides the code that translates between the device interface and the operating system, which requires a common interface for all similar devices.

The remainder of this chapter is organized as follows, we begin with a discussion of digital signals and their graphical display. We then introduce the concept of a bus – literally a time-shared signal path. We then introduce functions and latches and build a small memory from them. Finally we consider two examples of I/O devices – parallel I/O, and serial I/O. In the laboratory portion of Computer Structures, students learn to control a variety of hardware interfaces.

3.1 Signals and Timing Diagrams

In order to discuss the behavior of memory and I/O devices, we need a way to illustrate specific behaviors. A common way to do this for digital devices is a *timing diagram* which is essentially a (set of) graph(s) plotting the value versus time of one or more digital signals. A single signal may take three possible values – 0, 1, or Z where Z is an undriven value. In practice, a wire may be driven by more than one source (but not simultaneously) - in our diagrams, the case where no source is driving a signal is indicated by a Z state (shown as an intermediate state).

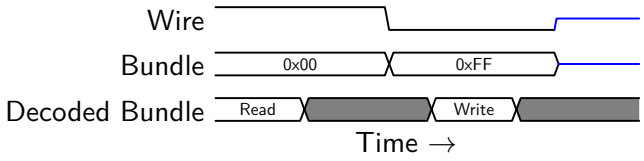


Figure 3.3: Example Timing Diagram

Consider the example timing diagram illustrated in Figure 3.3. There are three types of signals illustrated: a single wire, a bundle of wires treated as an 8-bit binary quantity, and a bundle for wires, with its binary value "decoded" based upon some encoding scheme. Consider the single wire which, over time, has three values – 1, 0, and finally Z. The 1 and 0 values are "driven" by a device while the Z value results from the associated device electrically disconnecting from the wire. The next signal is a bundle of wires where the current binary value is illustrated rather than showing the wires individually.

A bundle can also have the "value" Z. Finally, consider a decoded bundle where, at some times specific interpretations are illustrated (Read, Write) and at others the values are either unknown, or not germane to the discussion. All three of the signals illustrated are assumed to follow the same time base – when the decoded bundle has the value "write", the undecoded bundle has the value 0xFF and the single wire has the value 0. Finally, notice that the transitions between signal states are not instantaneous – there is a perceptible transition between stable states; this is a reflection of the reality of hardware – signals take time change.

Returning to the system diagram in Figure 3.1 and assuming static RAM (different types of memory have different low-level interfaces), a reasonable abstract timing diagram for a series of accesses to a single address is illustrated in Figure 3.4. This example timing diagram illustrates a sequence of three accesses to address 0x04 – read, write, read. The first read returns a previously stored value while the second read returns the value written – 2. This diagram hints at some of the underlying timing constraints. Notice that the data lines are in the Z state except when driven by the CPU (write) or memory (Read). Further notice that from the start of the read operation until the data lines are driven, time elapses. In addition, for a write operation, the data source continues driving the data lines until after the write ends. The diagram does not show which source is driving a signal – only the state of that signal over time. For real memories, the constraints that define the relationship between transitions on a timing diagram can be quite complex and define a contract between the designers of the various system components. In this chapter, we will gloss over such implementation dependencies.

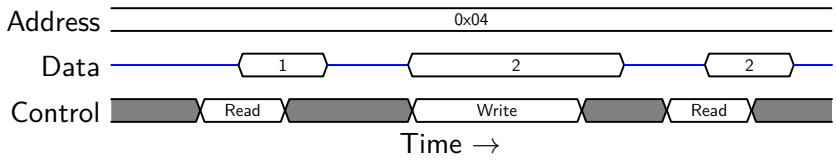


Figure 3.4: Example Memory Access

There are many more relationships between signals that such timing diagrams can illustrate. Frequently, they are used to illustrate specific real-time constraints and causal orderings each of which can be defined with additional annotations. The three states we have chosen to illustrate are not the only types of signal states that arise in real hardware. However, the concepts illus-

trated in this example should be sufficient for this chapter. For greater depth, you might consider a digital design textbook.

3.2 Functions and Latches

The study of digital systems design commonly is partitioned into two major topics – combinational logic and sequential logic. Loosely, combinational logic consists of functions built from simpler building blocks (for example 2-input logic functions), and sequential logic provides the state holding elements. In this section we introduce three basic types of components – functions, latches, and buffers. We assume that we can implement any reasonable function over groups of bits with simple operators and leave the study of such implementations to a digital design course. We introduce multi-bit latches that are capable of storing fixed-width numbers. Finally, we introduce multi-bit buffers which allow us to selectively drive a shared signal (e.g. the data lines in our generic system).

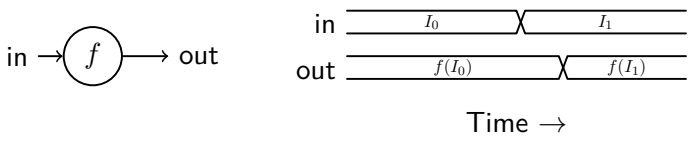


Figure 3.5: Function Component

Figure 3.5 illustrates a generic function component and its corresponding timing diagram. The function block computes an arbitrary function over its input(s). Although the example only has one input signal, we will define functions with multiple inputs. In general, this computation takes time – for example a *serial adder* requires time proportional to the size of its inputs. In the timing diagram, we illustrate this delay between changes in inputs and the resulting change of outputs with a visible, but arbitrary delay.

The other major building block for circuits is a latch. This is a component that captures and holds its input(s) when instructed to do so by a load (ld) signal. This component is illustrated in Figure 3.6. Notice that when the `ld` signal is high (*active*), the inputs are copied to the outputs and that when the `ld` signal is low, the outputs are preserved irrespective of input changes. The grayed out section of the timing diagram refers to an unknown value – in this case we do not know what value the latch has prior to the first active load. Notice that we have captured the property that output changes take

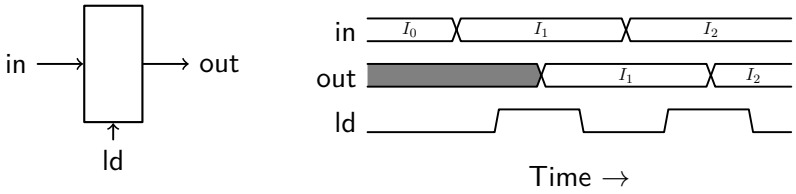


Figure 3.6: Latch Component

time to propagate. In a "real" circuit design we would also need to worry about the timing relationship between changes in input and the `ld` signal. In this context we are concerned only with the abstract behavior.

Given function blocks and latches we can build any finite-state machine – even a computer. In a digital design course you might learn how to do this using single-bit latches and 2-input logical functions.

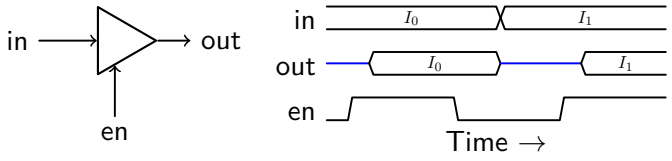


Figure 3.7: Buffer Component

The final building block that we consider is the buffer illustrated in Figure 3.7. If we were to consider a single bit buffer, its output could take three possible values – 0, 1, and Z. 0 and 1 correspond to Boolean values, while Z is an "undriven" value. We can safely connect the output of two buffers together, for example to share a set of data lines, so long as we guarantee at most one is *enabled* at any time. Notice in the corresponding timing diagram, that when the buffer is *enabled* (i.e. its `en` signal is high), the input of the buffer is reflected on its output – after a delay. When the buffer is not enabled, its output is not driven (it is in the Z state).

As an example, we can combine these three components to implement the single word memory illustrated in Figure 3.8. An N-bit latch is used as storage. The latch is loaded when the enable signal is valid; this occurs when the address bus has the value of a constant `A` and the control bus has the value `write`. The stored value in the latch is *driven* onto the data bus when

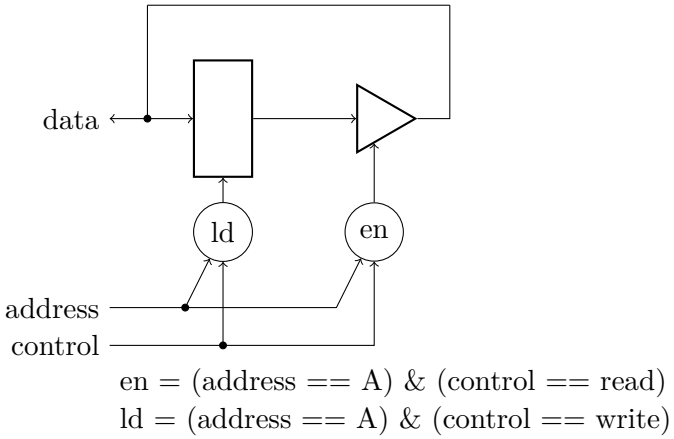


Figure 3.8: Single Word Memory

the address is `A` and control is `read`. A larger memory can be created by instantiating copies of this design with different values for constant `A`. "Real" memories are organized as arrays with much of the selection logic (ld,en) being reused across the array.

3.3 General Purpose I/O

A General Purpose I/O (GPIO) peripheral is provided with most microprocessors. Its function is to allow a program to read or write digital values provided by external circuits to the "pins" on the package of the microprocessor. In most cases, each pin is individually configured as an input or output. In the case where the pin is configured as an output, a program may change the value of the pin by writing to an *output register*. Similarly, if the pin is configured as an input, a program may read the value of the pin by reading an *input register*. The contents of the the input register are typically loaded at regular intervals with a separate sampling circuit. Finally, the pin direction is configured through a *control register*. A simple one-bit GPIO port is illustrated in Figure 3.9. As a simplification, we have elided all of the control signal generation, which is similar to that in Figure 3.8, but with separate control signals for each register. The input register is named `in`, the output register is named `out`, and the control register is named `dir`. An N-bit GPIO differs only in that each pin is connected to separate bits of N-bit input and output registers, and is configured by separate bits of an N-bit control register.

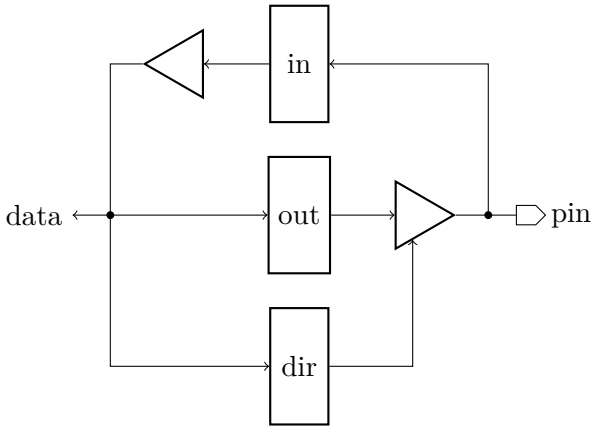


Figure 3.9: Single Bit GPIO

Suppose the addresses of the various latches are configured sequentially – we might describe this region of memory as a structure as in the following example, which illustrates the use of the various latches.

```
typedef struct {
       volatile unsigned int in;
       volatile unsigned int out;
       volatile unsigned int dir;
} port;

// examples
extern port *p;

p->dir = 3;   // set bits (pins) 0 & 1 to outputs
p->out = 1;   // turn on pin 0;
x = p->in;    // read the pin values
```

Assuming that all latches (registers) are initially zero, then all pins are initially configured be inputs. In order to use a pin for output, it is necessary to first write the corresponding bit in the direction register. We can then write the values on the output pins by writing to the output register (inputs are not affected) and read the values of the pins by reading the input registers.

Notice the use of the volatile keyword – when accessing hardware devices, this is an important hint to the compiler that these accesses should not be optimized away and the values should not be copied to temporary variables.

3.4 Serial I/O

The GPIO component of the previous section could be viewed as simply a set of latches controlling/accessing pins – writing to one latch changed the orientation of the pin, writing to another latch changed the value presented on a pin, while reading from a third read the current pin state. Most I/O devices have considerably more intelligence; in these, writing to a register causes the device to perform some action, rather than simply modifying stored state. In this section, we will describe a simple Serial I/O device, modeled upon the USART (universal synchronous asynchronous receiver transmitter) device in the STM32F303. The STM32F303 USART is a complex device that is configurable to perform a number of communication related functions; however, in this section we will focus upon a single one of these functions – asynchronous serial communication (UART).

Asynchronous serial communication in its most primitive form is implemented over a symmetric pair of wires connecting two devices – here we'll refer to them as the host and target, although those terms are arbitrary. Whenever the host has data to send to the target, it does so by sending an encoded bit stream over its transmit (TX) wire; this data is received by the target over its receive (RX) wire. Similarly, when the target has data to send to the host it transmits the encoded bit stream over its TX wire and this data is received by the host over its RX wire. This arrangement is illustrated in Figure 3.4. This mode of communications is called "asynchronous" because the host and target share no time reference. Instead, temporal properties are encoded in the bit stream by the transmitter and must be decoded by the receiver.

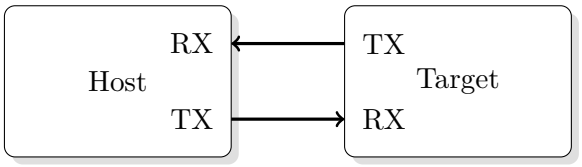


Figure 3.10: Basic Serial Communications Topology

A commonly used device for encoding and decoding such asynchronous bit streams is a Universal Asynchronous Receiver/Transmitter (UART) which converts data bytes provided by software into a sequence of individual bits and, conversely, converts such a sequence of bits into data bytes to be passed off to software. The STM32 processors include (up to) five such devices called USARTs (for universal synchronous/asynchronous receiver/transmitter) because

they support additional communication modes beyond basic asynchronous communications.

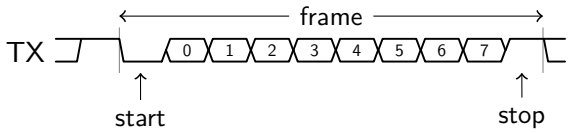


Figure 3.11: Serial Communications Protocol

One of the basic encodings used for asynchronous serial communications is illustrated in Figure 3.12. Every character is transmitted in a "frame" which begins with a (low) start bit followed by eight data bits and ends with a (high) stop bit. The data bits are encoded as high or low signals for (1) and (0), respectively. Between frames, an idle condition is signaled by transmitting a continuous high signal. Thus, every frame is guaranteed to begin with a high-low transition and to contain at least one low-high transition. Alternatives to this basic frame structure include different numbers of data bits (e.g. 9), a parity bit following the last data bit to enable error detection, and longer stop conditions.

There is no clock directly encoded in the signal (in contrast with signaling protocols such as Manchester encoding) – the start transition provides the only temporal information in the data stream. The transmitter and receiver each independently maintain clocks running at (a multiple of) an agreed frequency – commonly, and inaccurately, called the baud rate. These two clocks are not synchronized and are not guaranteed to be exactly the same frequency, but they must be close enough in frequency (better than 2%) to recover the data.

Thus, the protocol implemented by a UART is relatively complex – there are a lot of data and time-dependent transitions that must occur at a pin in order to transmit a single character. UART communication is also slow – in the time it takes to transmit a single character, a CPU may execute 10's of thousands of instructions. While it would be feasible to implement the UART protocol with GPIO, to do so would consume a tremendous fraction of the CPU capacity. Instead, we depend upon a complex peripheral to take care of the low-level details on behalf of the processor. In Figure 3.12 we illustrate a basic UART device. From the CPU's perspective, this UART appears as to be a set of four memory mapped registers. When the CPU has data to transmit, it writes to the transmit data register (TDR). When there

The register acronyms are chosen to match those used in the STM32.

is data to receive, the CPU reads it from the receive data register (RDR). The Because transmitting data takes time, the CPU must check if the TDR is empty before attempting to send a character – it does this by reading a status register (ISR). Similarly, it can read the status register to determine if there is a valid character in RDR. Finally, the UART protocol can be configured in a number of ways – baud rate, stop bits, etc. Thus, our model needs at least one control register (CTL) to perform this configuration.

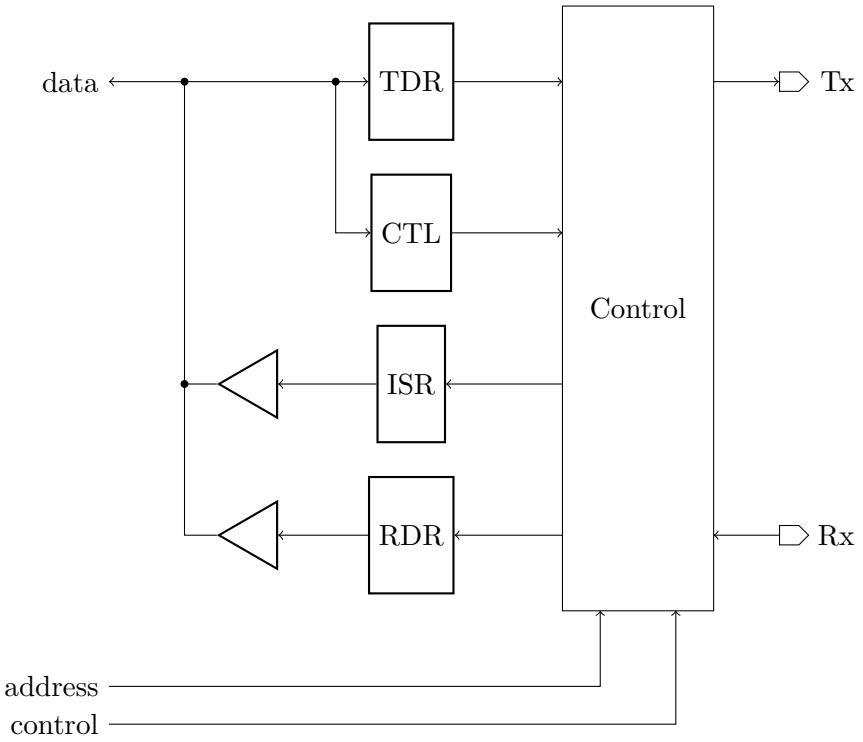


Figure 3.12: UART Device Model

In the general case there may be many memory-mapped registers associated with a single UART and there are typically several UARTs associated with each processor. It is common that the registers associated with a single device occupy a contiguous region of the address space. Thus, we can capture the device interface with a C structure definition. The following code illustrates a piece of the structure definition provided by ST for the STM32 processors (the STM32 USART has multiple control registers, which we have elided). The `__IO` type is equivalent to C `volatile`.

```
typedef struct
{
  ... configuration registers, etc.
__IO uint32_t ISR;    /* USART Interrupt and status register */
  ...
__IO uint16_t RDR;    /* USART Receive Data register         */
  ...
__IO uint16_t TDR;    /* USART Transmit Data register        */
  ...
} USART_TypeDef;
```

Given this interface, we can implement routines to send and receive data as follows. To send data, we must first wait until `TDR` is empty by reading the status register and checking the relevant bit (transmitter empty). Transmitting data is then a simple matter of writing to `TDR`. Similarly, to receive data, we must first wait until `RDR` is full (receiver not empty), and then read the available data from `RDR`.

```
void USART_SendData(USART_TypeDef* USARTx, uint16_t Data){
 while (!(USARTx->ISR & USART_FLAG_TXE));
 USARTx->TDR = (Data & (uint16_t)0x01FF);
}

uint16_t USART_ReceiveData(USART_TypeDef* USARTx){
  while (!(USARTx->ISR & USART_FLAG_RXNE));
  return (uint16_t)(USARTx->RDR & (uint16_t)0x01FF);
}
```

As written, these routines wait on the UART by *polling* the status register. More sophisticated device interfaces use interrupts to alert the CPU that the device needs to be serviced. If there is an operating system, these interrupts cause control to be transferred from a running program, to the operating system device driver.

3.5 Summary

In this chapter we have introduced the concept of memory-mapped I/O. We have shown how a few basic building blocks can be used to implement memory and, more importantly, how that memory interface can be used by I/O devices. We illustrated two fundamental types of I/O devices – general purpose I/O in which registers directly control processor pins, and Serial I/O, in which a separate device controller, accessed through memory-mapped registers, implements a complex communication protocol on behalf of the CPU.

Chapter 4

Data Representation

There are several data formats that we will consider in this book – signed and unsigned integers, characters, and instructions. While floating-point is an important data representation, and the actual data format is relatively easy to understand, the semantics of floating-point operations are quite complex and beyond the scope of this book. [1] In any case, most Cortex-M cores do not have hardware support for floating point operations and depend upon software libraries.

A significant difference between C and more modern high-level languages is the relatively impoverished set of mechanisms in C for representing data types as well as the weak semantics for built-in data types. For example, consider the example program Figure 4.1. In this example, the integer variable `i` is initialized to 200 and then multiplied by 300, 400, and 500. Finally, the (negative) result is printed.[2] The problem, as experienced C programmers will recognize, is that the correct result – 12,000,000,000 – is too large to represent in 32 bits, which is the word size of the (emulated) ARM processor used to execute this program. Python, executing on a processor with a similar word size, will produce the correct result. All modern processors support arbitrary precision integers, but not at a primitive level, and not without performance costs.

Some languages, such as Python, are designed to support arbitrarily large integers, while others, such as C and Java not only limited to the underlying machine word size, but "fail" silently when that word size is exceeded.

[1] The notorious FDIV bug cost Intel at least $475 million in 1995.

[2] Where possible, we use the qemu ARM emulator to execute programs compiled for the Cortex-M0 for our examples. The command-line command used to execute our example is displayed in this figure.

```
#include <stdio.h>

int mult(int i, int j) { return i*j; }

main() {
  int i = 200;
  i = mult(mult(mult(i,500),400), 300);
  printf("%d\n", i);

}
```

```
qemu-system-arm -cpu cortex-m3 -semihosting -kernel overflow-example.elf
-884901888
```

Figure 4.1: Arithmetic Overflow in C

4.1 Radix Number Systems

In Chapter 2, we discussed information containers words, and memory. In this and subsequent sections we will discuss the encoding of information – literally how to represent data with bits – as well as how to perform key operations on data in those encodings. We begin with one of the most important data types – cardinal numbers (non-negative integers). Most readers will have at least a passing familiarity with the binary representation of unsigned integers and at least some will have been introduced to signed representations.

Our goal is for you to not only learn about the key number encoding styles, but also properties of those representations. For example, most processors, even when operating on low-precision integers (e.g. shorts), convert these to word sized integers, perform the desired operation, and then convert the results back. This should raise a number of questions such as *how do we know the result is correct ?* Similarly, many processors perform subtraction using addition – *how does this work ?*

The starting point for our discussion is fixed radix number systems. While there are interesting examples of variable-radix systems (e.g. representations of time), those are outside the scope of this book.

You probably recall that a number written in decimal notation such as 1234 can be interpreted as an integer by the following function:

$$(1 * 1000) + (2 * 100) + (3 * 10) + (4 * 1)$$

or

$$(1 * 10^3) + (2 * 10^2) + (3 * 10^2) + (4 * 10^0)$$

Exercise 4.1: Largest radix-r number

Prove, using induction, that for radix-r, the largest number that can be represented with N digits is $r^N - 1$.

Recall that $x^0 = 1$.

We can define this more formally as a function that defines the value of a string of decimal digits $d_{N-1}, d_{N-2}, .., d_0$:

$$(d_{N-1}, .., d_0)_{10} = \sum_{i=0}^{N-1} d_i * 10^i$$

This function defines the value of a radix-10 string – the value of every digit is weighted by a power of 10 (the radix). This formula applies to any fixed radix system (with positive weights) simply by replacing 10 by the appropriate radix and limiting the range of digits. For example, binary numbers have two digit values (0, 1) and a radix of 2:

$$(d_{N-1}, .., d_0)_2 = \sum_{i=0}^{N-1} d_i * 2^i, d_i \in \{0..1\}$$

More generally:

$$(d_{N-1}, .., d_0)_r = \sum_{i=0}^{N-1} d_i * r^i, d_i \in \{0..r-1\}$$

We are primarily interested in radix-10 and radix-2, but we can prove some interesting facts about the general case:

$$\sum_{i=0}^{N-1} (r-1) * r^i \;=\; r^N - 1,\; r > 0$$

Thus the largest number we can represent with N digits of radix r is $r^N - 1$. (See Exercise 4.1.)

We are also interested in the relationship between various "string manipulations" and arithmetic operations. As you know from the decimal system,

multiplication by 10 is performed by adding a zero digit to the right, and division by 10 is performed by removing a digit from the right (the removed digit is then the *remainder*). This approach works for any fixed-radix system.

$$(d_{N-1}, .., d_0, 0)_r = (d_{N-1}, .., d_0)_r * r$$

and similarly

$$(d_{N-1}, .., d_0)_r / r = (d_{N-1}, .., d_1)_r + d_0/r$$

Thus multiplication and division by the radix can be accomplished simply by *shifting* digits. As we shall see, *shifting* – both left and right, is a primitive operation for many processors including the ARM Cortex-M; although, as we shall see, shifting is also used for non-numerical data – for example, extracting a field from a packed structure.

We frequently wish to convert between radix systems – in particular decimal and binary. Conversion to decimal is fairly simple – apply the formula. For example:

$$10101010$$

is interpreted as:

$$1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

which, if you know your powers of two, is:

$$128 + 32 + 8 + 2 = 170$$

There are more efficient algorithms for doing this conversion (Horner's rule), but we need to do so by hand so infrequently that it is hardly worth your effort to learn them. A better use of your time would be to memorize the powers of two up to 2^{10}.

The conversion from decimal to binary is somewhat more challenging. Where we used multiplication to convert from binary to decimal, we'll use repeated division by 2 to convert from decimal to binary. Dividing a number (the quotient) by 2 produces a remainder $r \in \{0, 1\}$ and a new quotient. By repeatedly dividing until the quotient is zero, we obtain the bits of the binary representation, but in reverse order.

Repeated Division (mod 2)				
Quotients	13	6	3	1
Remainders	1	0	1	1

$$(13)_{10} = (1101)_2$$

Suppose we wish to convert 126_{10} to binary.

Repeated Division (mod 2)							
Quotients	126	63	31	15	7	3	1
Remainders	0	1	1	1	1	1	1

$$(126)_{10} = (111\ 1110)_2$$

The procedure described above yields the minimum length binary number needed to represent a decimal number. In most cases, when we do such a conversion, our goal is to end up with a particular length number – typically 8,16,... bits. We can safely add 0 bits to the left without changing the value of a binary number (the situation will be somewhat different for signed numbers). Thus

$$(111\ 1110)_2 = (0111\ 1110)_2$$

The space in the binary string is simply to make it easier to perform a visual comparison.

There is another format that is frequently used in calculators – *binary coded decimal* (BCD). BCD stores each decimal digit as 4 bits using only 10 of the available 16 unique codes.

Radix Addition

Although we use machine words to represent non-numeric data, numeric data and the corresponding arithmetic operations remain one of the most important applications. We have previously seen how shift operations can be used for multiplication and division by powers of two.[3] In this section we discuss addition of radix numbers.

Adding binary numbers really isn't any different than adding decimal numbers – we proceed from right to left adding digits and propagating any

[3] General multiplication and division algorithms are outside the scope of this book.

> **Exercise 4.2: Carry Bits**
>
> Prove that for radix-r addition, the carry bits are always 0 or 1.

carry. However, processors work with fixed-sized numbers which leads to some interesting corner cases that are best understood by returning to first principles.

Suppose that we wish to add two decimal numbers 1234 and 5678. In grade school you learned to perform this operation as follows:

$$\begin{array}{cccccc} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{1} & \\ & & 1 & 2 & 3 & 4 \\ + & & 5 & 6 & 7 & 8 \\ \hline & & 6 & 9 & 1 & 2 \end{array}$$

The numbers in boxes are the *carry* bits – they will always be 0 or 1 for any fixed radix.

$$\begin{array}{cccccc} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \\ & & 1 & 0 & 1 & 0 \\ + & & 0 & 1 & 1 & 1 \\ \hline & & 0 & 0 & 0 & 1 \end{array}$$

In this example, the result is too large to represent in four bits, and there is a carry out, which would naturally constitute a 5th bit – the result 10001.

Remember, if there is a carry out from adding two N-digit positive numbers, then the result is too large to represent with N digits.

By capturing the carry out, we can perform arithmetic on large numbers through a series of smaller steps. Consider how we might perform the addition of two 8-bit numbers as two 4-bit additions:

$$\begin{array}{cccccccccc} & & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ + & & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline\hline & & & & & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \\ & & & & & \vdots & 1 & 0 & 1 & 0 \\ + & & & & & \vdots & 0 & 1 & 1 & 1 \\ \hline & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} & \vdots & & & & \\ & & 0 & 1 & 1 & 1 & & & & \\ + & & 0 & 0 & 0 & 1 & & & & \\ \hline & & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Later we will show how processors use this principle to implement multi-word arithmetic with instructions that perform arithmetic on words.

4.2 Subtraction Using Complements

In the previous section we discussed addition of fixed-radix numbers – in particular binary – and discussed the properties and role of carry bits. A similar approach can be used to perform subtraction – subtract individual digits from right to left while propagating a *borrow* bit. However, most digital hardware performs subtraction in a different manner known as the *method of complements* which performs subtraction by adding positive numbers. This technique can be used with any radix, but we will use it with radix-2. Later we'll see that the method of complements leads naturally to the most common binary representation for signed numbers – two's complement.

We perform the complement of a number of fixed radix r, by replacing each digit d_i by $(r-1)-d_i$. So for radix-10 we compute the complement by replacing 0 by 9, 1 by 8, ... 9 by 0. More precisely, this is called the nine's complement – we replace every digit d_i by $9-d_i$.

To subtract decimal number X from Y, we form the nine's complement of X, $\overline{X}$, and add $\overline{X}+1$ to Y. For example, to subtract 33 from 45:

$$\begin{array}{cccc} \boxed{1} & \boxed{1} & & \\ & 4 & 5 & \\ & 6 & 6 & \\ + & & 1 & \\ \hline & 1 & 2 & \end{array}$$

We discard the final carry out to obtain our result.

Where $\overline{X}_{10}$ is called the nine's complement, $\overline{X}_{10}+1$ is called the ten's complement. In the case of radix 2, the number formed by complementing the digits is called the one's complement; the one's complement of a number plus one is called the two's complement.

Suppose we wish to subtract 00111 from 10001 using the method of

complements.

$$\begin{array}{cccccc} & 1 & 0 & 0 & 0 & 1 \\ - & 0 & 0 & 1 & 1 & 1 \\ \hline\hline \\ \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} & \\ & 1 & 0 & 0 & 0 & 1 \\ & 1 & 1 & 0 & 0 & 0 \\ + & & & & & 1 \\ \hline & 0 & 1 & 0 & 1 & 0 \end{array}$$

To understand why this works, consider the general formula for fixed radix numbers. For radix r, the r-complement is computed by complementing each digit and adding one to the result.

$$\sum_{i=0}^{N-1} d_i * r^i$$

Replace each digit by its complement and add 1:

$$\sum_{i=0}^{N-1} (r - 1 - d_i) * r^i + 1$$

We can refactor this:

$$(\sum_{i=0}^{N-1} (r-1) * r^i) + 1 - (\sum_{i=0}^{N-1} d_i * r^i)$$

or $(r^N - 1) + 1 - (\sum_{i=0}^{N-1} d_i * r^i)$

Which is $r^N - (\sum_{i=0}^{N-1} d_i * r^i)$

In the case of fixed-precision, r^N is the carry out which we discard.

4.3 Negative Numbers

In the preceding section we discussed arithmetic with non-negative integers. In this section we discuss methods for encoding negative integers. There are three common techniques – sign magnitude, biased, and two's complement. Only the last of these is routinely used for integer representation, but the other two are used in the representation of floating-point numbers.

The most obvious way to represent signed integers is to duplicate the technique that we use when writing decimal integers – use a dedicated sign bit.

Suppose that we to represent -13 as an 8-bit number. Recall that $(13)_{10} = (1101)_2$; using the 8th bit as a sign bit, where 1 is interpreted as negative:

$$-(13)_{10} = (10001101)_{sm}$$

We can formally define signed magnitude as:

$$-1^{d_{N-1}} * \sum_{i=0}^{N-2} d_i * 2^i$$

There are two problems with this representation. The first, that there are two zeros (+0 and -0) is relatively minor, but could complicate operations like comparison. The second is more substantial. Basic arithmetic involves a number of special cases – for example, adding two numbers with different signs, adding two negative numbers, or worse, handling integers with differing numbers of bits. Processor designers abhor special cases because they take additional time on the critical path.

Biased numbers are defined by subtracting a constant from an unsigned representation. For example,

$$\sum_{i=0}^{N-1} d_i * 2^i - 2^{N-1}$$

In the IEEE floating point number representation, biased numbers are used for exponents (which can be negative for fractions).

It turns out that there is a more natural representation that requires no special cases, can deal with both signed and unsigned representation, and lends itself naturally to working with numbers of differing precision – this the two's complement form. Recall that we can form the two's complement of a binary number by inverting its bits and adding 1. When we did this for subtraction, we glossed over tracking the sign bit. To define the value of a two's complement number, we need some way to capture this information; we do so using a negative weight:

$$(b_{N-1}..b_0)_{\overline{2}} \;=\; -b_{N-1} * 2^{N-1} + \sum_{i=0}^{N-2} b_i * 2^i$$

Here are a few 8-bit examples:

Exercise 4.3: Two's Complement Number Range

Given the formal definition, derive the minimum and maximum two's complement numbers that can be represented in N bits.

Exercise 4.4: Two's Complement Operation

For a number B with magnitude less than 2^{N-2}, show that if B is represented by a 2's complement number with N bits $b_{N-1}..b_0$ then

$$-(b_{N-1}..b_0)_{\overline{2}} = (\overline{b_{N-1}..b_0})_{\overline{2}} + 1$$

- 0000 0000 = 0 (unique)
- 0000 1010 = +ten
- 1111 0110 = -ten
- 0111 1111 = +127 (largest)
- 1000 0000 = −128 (smallest)

- 0x0000 0000 = 0
- 0x0000 00AA = +170
- 0xFFFF FF56 = -170
- 0xFFFF FFFF = -1
- 0x7FFF FFFF = 2,147,483,647 (`INT_MAX`)
- 0x8000 0000 = -2,147,483,648 (`INT_MIN`)

`INT_MAX` and `INT_MIN` are defined in `<limits.h>`

One of the beautiful things about two's complement is that addition is performed in exactly the same manner as unsigned addition – at least for fixed bit-width.

- Two's complement numbers are:

Exercise 4.5: Sign Extension

Prove that "sign-extension" is value preserving.

  - **added** in the same way as unsigned numbers.
  - **negated** by the two's complement operation.
  - **subtracted** by negating the subtrahend and adding.

- Overflow is more complicated.
  - Addition of numbers of opposite sign, cannot result in overflow.
  - Addition (subtraction) of numbers of the same sign causes an overflow if the "sign" bit is different than that of the operands.

Two's complement numbers have some other important properties. We can always determine the sign by looking at the *most significant digit*; i.e. to digit with the highest weight. As a convenience, we call this the sign bit.

An N-bit two's complement number can be converted to an N+1 bit number with the same value by duplicating the sign-bit – this is called sign extension. Similarly, an N-bit two's complement number can converted to an N-1 bit number by dropping the sign bit if the new sign bit has the same value.

Formally,

$$(d_{n-1}d_{n-2}...d_0)_{\overline{2}} = (d_{n-1}d_{n-1}d_{n-2}...d_0)_{\overline{2}}$$

Sign-extension is important, because is provides a natural way to cast a two's complement number to a larger value.

We can multiply a two's complement number by two by shifting left one bit:

$$(b_{n-1}..b_0)_{\overline{2}} * 2 \;=\; (b_{n-1}..b_0, 0)_{\overline{2}}$$

If $b_{n-1} = b_{n-2}$ then we can drop the extra sign bit.

We can divide a two's complement number by two by shifting right and duplicating the sign bit.

$$(b_{n-1}..b_0)_{\overline{2}}/2 \;=\; (b_{n-1}, b_{n-1}..b_1)_{\overline{2}}$$

This right shift operation is called "arithmetic shift right."

Exercise 4.6: Shift Operations

Prove that left-shift is equivalent to multiplication by two and arithmetic right-shift is equivalent to division by two.

4.4 Characters

For many years, the most common character set for computer programs was ASCII, which stands for "the American Standard Code for Information Interchange. ASCII was designed to encode the English alphabet and has 128 7-bit characters including 95 that are printable. There are a number of non-printable characters including "bell" (7) which specifies that a bell should be rung as well as characters that functioned for transmission control. ASCII dates to the early 1960's and was initially used for teleprinters.

In C ASCII characters are stored as 8-bit quantities. The `char` data type is a signed 8-bit integer, but all ASCII characters are non-negative. A few ASCII characters are notable. The ASCII NUL character is 0 – recall that all C-strings are 0 or NULL terminated. The alphabetic characters A-Z are represented by numbers 65-90; a-z are represented by 97-122. Finally the digits 0-9 are represented by 48-57. These encodings make sorting and case-converting strings easy.

The C language standard does not require that characters be encoded in ASCII form. The standard does require:

- 0 is the NULL character.
- The character set must include the 26 uppercase and 26 lowercase letters of the Latin alphabet.
- The character set must include the 10 decimal digits – these must be encoded with successive numbers starting from 0.

The standard goes on to specify a minimum set of 29 graphic characters and a handful of control characters.

The ASCII character set is obviously a problem with languages other than English. Starting in the 1980's a "universal character set", called Unicode was developed. There are actually a number of Unicode standards, including UTF-16 and UTF-8. UTF-8 is especially notable because it is a variable length code that includes all of the ASCII characters as its first 127 characters.

Furthermore ASCII bytes do not occur in non-ASCII code, thus UTF-8 is safe to use whatever ASCII characters are used.

4.5 C Integral Types

The integral types in C are the basic data building blocks; these include signed and unsigned integers as well as floating point numbers. All of these types are implementation dependent; however, all implementations must satisfy some basic ranges. For example, consider C specification for the (minimum) ranges of various signed integer types:

$$\begin{array}{ccccccccc} 2^{63} & & 2^{31} & & 2^{15} & & 2^{15} & & 2^{7} \\ \vee & & \vee & & \vee & & \vee & & \vee \\ \text{long long} & \supseteq & \text{long} & \supseteq & \text{int} & \supseteq & \text{short} & \supseteq & \text{char} \\ \vee & & \vee & & \vee & & \vee & & \vee \\ -2^{63} & & -2^{31} & & -2^{15} & & -2^{15} & & -2^{7} \end{array}$$

These range requirements can be satisfied with either 2's complement or sign-magnitude representations of 64 (long long), 32 (long), 16 (short), and 8 bits (char). Notice that the C specification also imposes an ordering on the various types; for example, all long values are representable as long long.

The fact that the C specification leaves so much freedom to the implementation is both a blessing and a curse. The freedom allows the creation of efficient C implementations for processors with wildly varying resources – from tiny 8-bit microprocessors to large 64-bit workstation processors. However, the variation between implementations greatly complicates the task of writing portable code. The C specification dictates standard header files that make it possible for a running program to easily determine the ranges of various types – `<limits.h>`. It has been common, where code portability is important to use integer types that explicitly define their size: `int8_t`, `int16_t`, ... These are defined in `<stdint.h>`.

The floating-point times form a similar order. In practice float is generally implemented with 32-bit IEEE 754 specification and double with 64-bit IEEE 754; although, C specifications are limited to defining required properties of these numbers. The interface requirements for the floating point types are defined in `<float.h>`.

There are some other specialized types, for example, `size_t` is the container for the size of an arbitrary C object (for example and array). This is often, but not always, the same size as an unsigned integer or an unsigned long long.

Type Promotion/Conversion

In addition to defining the properties of the basic types, C defines how types are promoted during execution. For example, char and short operands are generally promoted to int for arithmetic operations and then converted back to char or short *when the computation is complete*. Exactly when this down-conversion occurs is a bit nebulous. One might assume it occurs when a result is written to a named variable, but in practice this might not occur until after a serious of subsequent computations. This isn't an issue as long as the intermediate results can be represented with the smaller type. However, there are no such guarantees in C.

When performing operations with integers of differing size, C specifies that the smaller variable should be converted to the larger size – for example int to long long. Where computations occur between signed and unsigned values, the unsigned value is converted to signed – again trouble occurs when the an unsigned value cannot be represented in the signed type.

Constants

hex, long, long long, ...

Chapter 5

Stored Program Interpreter

Through this and subsequent chapters we will introduce the fundamental idea of a computer as the interpreter of a relatively impoverished language (machine instructions) and the translation of C into this language. Throughout this presentation we will use the ARM Thumb instruction set as a running example – specifically the subset defined for the Cortex-M0 processors. The approach that we will take is to begin with a "stripped" model and, gradually build up to the complete Cortex-M0 instruction set. Through a series of programming exercises, you will be asked to write an interpreter for this instruction set. At each stage we will provide the scaffolding necessary for you to write and test your interpreter on the current instruction (sub)set. You will use the gnu-arm tool-chain to write test programs, the qemu-arm simulator to provide a reference interpreter, and the gdb debugger to assist in developing your test programs. The final destination of this series of exercises will be an interpreter, written in C, that can load and execute Cortex-M0 binaries compiled from C and assembly code using the gnu-arm compiler tool-chain.

Our educational objectives are that at the end of this process you understand the following:

1. How the ARM Thumb instruction set is interpreted by a processor.
2. The C memory model; i.e. how the processor resources are used by C programs to store and access data.
3. How C programs can be translated to the ARM Thumb instruction set.
4. The ARM ABI (application binary interface) which dictates how "legal" programs cooperate to use the machine resources.

The ARM Thumb instruction set is widely used in the embedded variants of the ARM 32-bit processor, which is the most widely used 32-bit processor (family) on the planet. In the original ARM processors, all instructions were 32-bits (4-bytes); subsequently, a 16-bit variant (the Thumb instruction set) was introduced in order to improve memory utilization. Initially, processors supporting Thumb instructions could switch (at the procedure level) between Thumb and 32-bit instruction sets. More recently, ARM has introduced a family of 'M' processors that execute only Thumb instructions (plus a very limited set of 32-bit instructions). These 'M' processors include the M4, M3, M1, and M0. Of these, the M0 has the smallest set of instructions. It is notable, that every ARM processor is capable of executing programs written with the Thumb-M0 instruction subset. For embedded applications, processors based on the ARM 'M' core provide both high-performance and low-cost; it is possible to purchase single Cortex-M0 processors for a few dollars.

5.1 The Stored Program Model

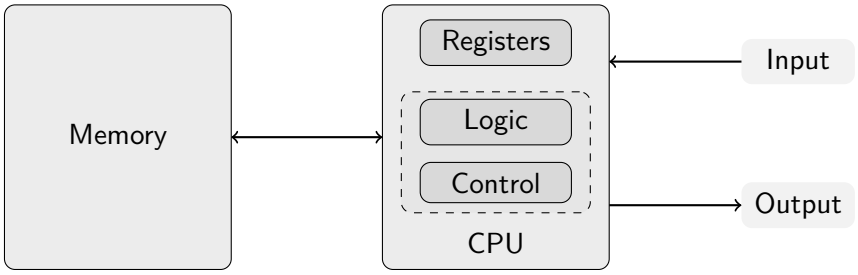


Figure 5.1: Stored Program Machine (von Neumann architecture)

In Figure 5.1 we illustrate a simple "stored program" computer model. The model consists of three major components – memory, which is used to store instructions (programs) and data; a CPU (central processor unit), which provides programmer visible resources in the form of registers, and a program interpreter consisting of logic and control hardware; and some mechanisms for input and output. The Cortex-M0 interpreter that we will use to explore the ARM instruction set follows this model except that our interpreter program fills the role of the Logic/Control provided as hardware in a real processor. Our development will largely ignore the Input and Output devices – this is the subject of a separate laboratory.

The illustrated architecture is more commonly referred to as a "von Neumann" architecture in honor of John von Neumann who described such a machine in 1945. A central idea of this architecture is that memory is used to store both data and *instructions*. During execution, the CPU fetches (reads) instructions from memory, interprets them, and, if necessary, writes results to memory. The basic instruction execution cycle is illustrated by the code fragment in Figure 5.2.

```
extern inst_t M[];          // the memory
extern unsigned int pc;    // index into the memory

while (1) {
      inst_t inst;   // an instruction
      inst = M[pc++];
      interpret(inst);
}
```

Figure 5.2: Instruction Interpreter (Fragment)

In this code fragment, memory is treated as an array of instructions addressed by a dedicated *program counter* register – `pc`. Individual instructions are read from memory into a private variable and then interpreted. Interpretation consists of extracting bits from the instruction and using these to determine what operation to perform on which operands. For example, the Thumb instruction:

```
adds r1, r0
```

is interpreted as:

```
r1 = r1 + r0
```

`r0` and `r1` are examples of *registers*, which are dedicated programmer-visible temporary storage. The subset of the Cortex-M0 that we consider has eight such registers named `r0` – `r7`. The textual form of the instruction shown above is *encoded* as the 16-bit word `0x1C41`. Eventually, we shall examine how instructions are encoded.

The von Neumann architecture has a major performance bottleneck – the single pathway between memory and the CPU. Memory accesses take a relatively long time (they have a high *latency*) compared to instruction interpretation yet a single instruction in general-purpose processor may need to access (reference) memory multiple times. Programmer visible registers can alleviate this bottleneck because registers can be used to temporarily

store values in use. Another instance of this bottleneck is the *bandwidth* (i.e. the number of bits per unit time) required simply to fetch instructions. The Thumb instruction set is relatively efficient in this respect because two 16-bit instructions can be accessed in a single memory operation. Registers reduce the memory bandwidth required for naming (addressing) values. For example, accessing a value in memory requires a 32-bit address, while accessing a Thumb register requires a 3-bit address (registers are numbered 0-7). Indeed a single 16-bit Thumb instruction can reference as many as three registers (two in most instructions). In general, memory bandwidth and latency are two of the most significant performance constraints for a processor. Briefly, bandwidth is the amount of data that can be written to or read from memory per unit time, while latency is the time required to access a random memory location.

5.2 The ARM Thumb Processor Model

In the ARM family of processors, memory is defined as an array of bytes and may contain data, addresses, and instructions. In the ARM processors, memory can be accessed as bytes, half-words (2 bytes), and words (4 bytes), with some constraints. For example word-sized items are always *aligned* on 4-byte boundaries. The Thumb instructions are mostly 2-bytes with a handful of 4-byte instructions. Addresses are always 4-bytes. Finally, data may be 1, 2, or 4 bytes. The lines between data, instructions, and addresses are not always well defined.

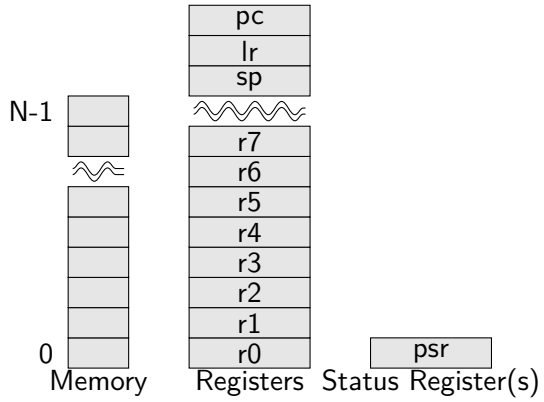


Figure 5.3: Programmer Resources of Cortex-M0

The ARM architecture also includes registers `r8-r12`, but these are not accessible from most Thumb instructions and hence we will ignore them for the moment. Registers `sp`, `lr`, and `pc` are also named `r13`, `r14`, and `r15`, respectively; however, we will refer to them by their *special purpose* names (`sp`, `lr`, and `sp`).

The 32-bit ARM instructions can directly access registers `r0-r16`

The ARM processors support three basic types of primitive operations (instructions): Data Processing, Memory Reference, and Control Flow. The `add` instruction is an example of a data processing instruction. Data processing instructions include arithmetic operations such as addition, subtraction, and multiplication as well as *logical* operations such as and, or, and not. Data procession instructions operate on data stored in registers as well as *constants*, which are small signed or unsigned integers (depending upon the instruction). For example,

```
adds r1, 4
```

adds the constant "4" to register `r1`.

An important, but subtle point is that data processing instructions operate on anything represented as a 32-bit word of bits – this includes program data, program instructions, and, addresses. For example, in order to access a field within a structure, we need the address of (pointer to) the structure; we then compute the address of the field by adding an appropriate offset (constant) to the structure pointer. Finally, we use a memory reference instruction to access the contents of the structure field. While this example includes a lot of ideas to be discussed in this and subsequent chapters, the important point to remember is that complex operations in a programming language must be translated into a small set of primitive operations at the machine level in order to be executed.

The second category of instruction is the *memory reference* instruction. This includes operations to load from and store to locations in memory. Given that there are only 8 programmer visible registers, most program data is stored in memory. In order to operate on this data, it must first be loaded into a register; once an operation is complete, the result is written back to memory.

For a C programmer, the best way to think of memory reference instructions is as pointer operations. Here are example load (`ldr`) and store (`str`), which read from and write to memory.

```
ldr r0, [r1]          @   r0 = *((uint32_t *) r1)
str r0, [r1]          @   *((uint32_t *) r1) = r0
```

There are variants of these two operations for loading and storing byte and half-word values. There are also variants that simplify the address calculation; for example, to make accessing structure fields more efficient. Some of these additional addressing modes will prove to be essential.

The final category of instructions are *control-flow* instructions. As defined in Figure 5.2, instructions are fetched and executed in strictly straight-line order. In order to implement a programming language we need a mechanism to jump over blocks of code (for example in an "if-then" or conditional construction), jump to random locations (e.g. a procedure) and return from a procedure call. Although a language such as C has many control-flow operations (e.g. for, while, do, continue, break, if, then), they can all be implemented with a small number of primitive operations. Here we present a single example. Consider the C fragment which tests if a variable (conveniently named r0) is "true" and, conditionally, jumps to the code at label1.

```
   if (r0)
      goto label1;
   ...
label1:
```

This can be implemented in assembly language with a *compare* instruction, which compares `r0` with 0, and a *conditional branch* instruction which performs the jump – in this case if `r0` is *not equal* (ne) to 0.

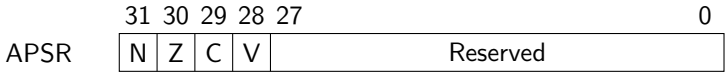
```
   cmp r0, 0
   bne label1
   ...
label1:
   ...
```

In Chapter 7, we show how all C language conditional operations can be implemented with this basic pattern.

Status Register

The preceding example raises an interesting question – *how does the branch instruction "know" that the result of the compare instruction was zero ?*. In the ARM processor, as with most processors, instructions that manipulate data *may* modify the status register as a side effect. The Cortex-M0 has a status register (APSR)[1] which has four bits to capture this information:

[1]Technically, the APSR is one of three 32-bit registers that comprise the PSR illustrated in Figure 5.3. The other registers functions outside the scope of this discussion.



N Negative – this bit is set when the result of an operation is negative (i.e. bit 31 is 1).

Z Zero – this bit is set when the result of an operation is zero.

C Carry – this bit is set when an (arithmetic) operation has a carry out.

V oVerflow – this bit is set when the result of an arithmetic operation has the wrong sign and therefore does not fit in 32 bits. For example, the addition of two positive numbers yielding a negative result. Only occurs when adding numbers with the same sign or subtracting numbers with opposite sign.

The compare operation in the previous example is implemented using subtraction (`r0 - 0`) and discarding the result, but keeping the side effect – setting the four condition flags.

The names of these four condition flags are universally understood – virtually every processor implements them and the semantics are fairly standard. [2] When reading the definitions of the various instructions, it is important to note which condition flags are set by an instruction and under what conditions. In the full ARM instruction set, condition flags are optionally affected – each instruction has a bit that determines whether or not the condition flags are set. In the thumb instructions used in the Cortex-M0, most data processing instructions affect the condition flags. A few, such as the arithmetic instructions, have optional control.

One notable aspect of C is that every expression returns a value that can be used as a condition – 0 is interpreted as false and anything else as true. Semantically,

```
if (expr) ...
```

is the same as

```
if (expr != 0) ...
```

[2] Some RISC processors – MIPS, DEC Alpha did not have central status flags, but rather captured the results of comparisons in general purpose registers.

When implementing a C expression in ARM assembly, we get this behavior without the need to explicitly compare the expression result to 0 – that computation is a side effect of the expression evaluation.

A second use of the condition flags is when performing multi-word arithmetic. The Cortex-M0 arithmetic instructions operate on 32-bit quantities, yet the instruction set can support addition and subtraction with larger quantities. To do this, the intermediate carry (C) flag is captured and used as a carry input for succeeding operations as in:

```
adds r0, r1
adcs r2, r3
```

which has the effect of adding two 64-bit numbers `(r2,r0)` and `(r3,r1)`. The first addition adds `r0` and `r1` (placing the result in `r0`) and captures the carry out (the "`s`" in `adds` means to set the condition flags). The second "add with carry" operation adds `r2` and `r3` capturing the result in `r2`. Effectively performing

```
(r2,r0) = (r2,r0) + (r3,r1)
```

While it is possible to read and write the APSR directly using special instructions, this is rarely needed outside of operating system code.

To summarize, the ARM processor has four condition flags that are set as a side effect of most data processing instructions – *you should memorize the names and meanings of these!* These condition flags are written by data processing instructions and read by control-flow instructions, and when performing multi-word arithmetic.

Instruction Encoding

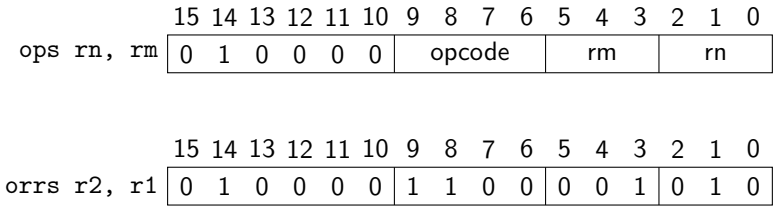


Figure 5.4: Data Processing Instruction Format

The instruction interpreter of Figure 5.2 operates on binary data – the instruction. Every assembly language instruction has a binary encoding; for

the Cortex-M0, these are mostly 16-bit words. As an example consider the bitwise logical or instruction:

```
orrs r2, r1   @  r2 = r2 | r1
```

`orrs` is one of 16 data-processing instructions that share a common format. This format and the corresponding encoding are illustrated in Figure 5.4. Notice that the format has specific values for bits 15-10 – the instruction interpreter (processor) uses these to determine the *class* of instruction and, in particular, to determine the operands needed by the instruction. For example all of these data processing instructions are of the form:

```
ops  rn, rm
```

Other instructions sharing this format are:

opcode	Instruction	Description
0000	ands	Bitwise and
0001	eors	Bitwise exclusive-or
0010	lsls	Logical shift-left
0011	lsrs	Logical shift-right
0100	asrs	Arithmetic shift-right
0101	adcs	Add with carry
0110	sbcs	Subtract with carry
0111	rors	Rotate right
1000	tst	Test
1001	rsbs	Reverse subtract (rsbs rn,rm,0)
1010	cmp	Compare
1011	cmn	Compare negative
1100	orrs	Logical or
1101	muls	Multiply
1110	bics	Bit clear
1111	mvns	Move negative

Notice that the instruction (as illustrated in the Figure) consists of the fixed format bits (010000), the orr opcode (1110), rm=r1 (001), and rn=r2 (010). Tthe only tricky part is being sure of the operand order.

It is important to remember that instruction encodings are designed to make hardware execution efficient, not for ease of readability. In general, you should become familiar with the various formats in order to understand possible restrictions on instruction use. A good example of such a restriction is the range of immediate constants – with 16-bit instructions these vary a lot between formats and are generally quite restrictive.

Exercise 5.1: Instruction Decoding

Write a program to decode the ARM thumb data processing instruction format (Figure 5.4) The prompted input (i.e., the user will be prompted to enter input while your compiled binary is running) to your program will be a list of hexadecimal strings such as:

```
4008
4050
4098
40E0
```

Your program is to read such hex and generate legal Thumb-2 assembly:

```
.text
.syntax unified
.thumb
   ands r0,r1
   eors r0,r2
   lsls r0,r3
   lsrs r0,r4
```

You are to decode only this one format. Hex input coresponsding to any other input should be printed as

```
.hword 0x....
```

Your binary should first display the 3 lines of "." information, then for each set of 4 hex values entered, it should show the following output. For example, once your program is running, if you enter `4008` at the prompt, the output should be `adds r0, r1`

The following small data set is provided for you to begin your work:

```
0:   4008   ands     r0, r1
2:   4050   eors     r0, r2
4:   4098   lsls     r0, r3
6:   40e0   lsrs     r0, r4
8:   4128   asrs     r0, r5
```

Hints For Exercise

You can easily generate more test cases by writing legal Thumb-2 assembly instructions in a file (e.g.)

```
.text
.syntax unified
.thumb
    ands r0,r1
    eors r0,r2
    lsls r0,r3
    lsrs r0,r4
    asrs r0,r5
```

and executing the following commands:

```
    arm-none-eabi-as test_case_file -o test.o
    arm-none-eabi-objcopy test.o -O binary --only-section=.text
        ↪test.bin
    hexdump -x test.bin |sed -e 's/^[^ ]*//' > test.hex
```

The file `test.hex` contains the hex encoded instructions. Remember to test your code thoroughly – a good test is to "roundtrip" a test file – generate the assembly, and use the instructions above to generate a hex file, and then use your program to generate a new assembly file. Your inputs and outputs should agree.

Consider structureing your code around a small set of "lookup" tables. For exmaple, you might build a table of register names:

```
char *regnames[] = {``r0'', ``r1'', ... ``pc''};
```

Similarly, you can build a table for opcodes:

```
char *opname[]   = {``ands'', ``eors'', ... };
```

Write macros to assist in extracting the various bit fields:

```
Rm(x) (((x) >> 3) & 0x3)
```

Readking the hex input with scanf is easy

```
while (scanf(``%x'', &inst) == 1){ ... }
```

5.3 Pipelining

Chapter 6

Data Processing

The C language has a rich set of operations from which arithmetic and logical expressions may be built. In this chapter, we discuss how C expressions can be realized with the Cortex-M0 instruction set. We begin with a review of the C operators and then show how complex C expressions can be reduced to simpler expressions that are more readily translated into assembly language. We focus exclusively on integer and binary data – as we have mentioned, floating-point is beyond the scope of this book.

Expressions in C may be used in assignment statements – for example,

```
x = y + 3;
```

and as conditions in loops and conditional constructs – for example,

```
if (x > 3) {...}
```

In the later case, the result of executing the expression is not saved, and only the "truth" value of the result matters. In both cases, expressions may refer to C variables. In this chapter, we do not consider the issue of how variables are referenced, rather we assume a small set of variables, the CPU registers, and show how expressions can be implemented with these. The general case of accessing C variables in memory as well as the address and indirection operators are postponed to Chapter 8 where we consider the topic of memory and memory addressing. We also postpone the question of dynamically allocating temporary variables (e.g. for evaluating complex expressions) until Chapter 9 where we discuss stack allocation. In short, we will be asking you to take some things on faith !

We assume that you understand that complex expressions can be evaluated as a sequence of simpler expressions; for example,

```
x = ((x + y + z) * 7 + (x + y + 3));
```

can be rewritten as a series of two-operand operations by introducing a temporary variable `t0`:

```
x  = x + y;
t0 = x;
t0 = t0 + 3;  // (x + y + 3)
x = x + z;
x = x * 7;    // (x + y + z) * 7
x = x + t0;
```

By a two-operand operation we mean one that reads and writes at most two operands which may be a combination of variables and constants. The C language specifies the *binding* order for all operators, so the conversion of complex expressions into a series of simpler ones can be done unambiguously.

Expressions in C may include operands of multiple types; for example `unsigned short` and `int`. Briefly, `char` and `short` operands are always promoted to `int` or `unsigned int` before any computation. These promotions are supported by a few data operations discussed in this chapter and memory operations discussed in Chapter 8. In general, with two integer operands of different sizes, the smaller sized operand is promoted to the larger type. Operations mixing signed and unsigned types have somewhat complex rules. Frequently, these conversions are performed using code sequences generated by the compiler. In this chapter we will assume operands that are signed and unsigned integers.

The C language has several operators that have side effects, notably the pre- and post- increment and decrement operators `++` and `--`. For example,

```
x = 3 + y++;
```

has the following effect

```
x = 3 + y;
y = y + 1;
```

These operations with side-effects exist purely as *syntactic sugar* for the programmer and can always be eliminated by rewriting the expression as a series of simpler expressions. Hence we ignore these.

Similarly, the various assignment operators – `+=`,`-=` – are easily translated into more conventional assignment statements as in

```
x = x + y;
```

instead of

```
x += y;
```

C also includes conditional expressions of the form

```
exp ? val1 : val0
```

These can always be converted into an `if` statement were the result is stored in a temporary variable:

```
if (exp)
    t0 = val1;
else
    t0 = val0;
```

The C language includes some operators that have no direct implementation in the Cortex-M0 assembly language – for example, division. Generally, C compilers provide small procedure libraries for operations that are not supported natively. For GCC, this library is called `libgcc` – there are different binary implementations for every significant variant of the ARM family. Rather than examine this topic in detail, we will focus upon the operations that are natively provided by the Cortex-M0 instruction set, and point out some that are not. Similarly, the Cortex-M0 instruction set is restricted to 32-bit data operations, while C supports 64-bit quantities. 64-bit operations are supported by the compiler through `libgcc`. The Cortex-M0 instruction set does provide some support for multi-word arithmetic, which we discuss. Finally, we postpone all discussion of the pointer operations `*` and `&` as well as pointer arithmetic to Chapter 8 where we discuss the broader topic of memory.

In the remainder of this chapter, we show how the various C operators are implemented in the Cortex-M0 instruction set. Our presentation is organized around the following categories of C operations: bitwise logic, shifting, arithmetic. We postpone two categories of operators – relational, and logical – to Chapter 7 where they can be presented in a more natural context. Throughout this presentation the operands that are used with these operators are severely constrained. For example, we have only a limited set of machine registers to serve as variables, and we will be limited to a small set of constants. Both of these restrictions will be eliminated in Chapter 8 where we discuss the use of memory for variable and constant storage.

6.1 C Type Conversion Rules

The C type conversion rules dictate how operands are handled prior to their use in expressions. These are all defined in the C standard [**?**]. Throughout this discussion, we will assume two-operand expressions.

Both `char` (`unsigned char`) and `short` (`unsigned short`) types are always promoted to `int` (`unsigned int`) before any operation is performed. These promotions are temporary – they do not change the storage size of the operand.

In the absence of floating point operands, which we ignore in this book. There are four cases to consider after the automatic promotion described above.

1. Both operands are of the same type.
2. Both operands are signed, but of different sizes.
3. Both operands are unsigned, but of different sizes.
4. One operand is signed and the other unsigned.

.

In case 1, the operation proceeds. In cases 2 and 3, the operand with the smaller size is promoted to the size of the other operands and then case 1 applies. Case 4, is complicated; if the two types are of the same size then the signed operand is converted to unsigned. Otherwise, the smaller operand is converted to the type of the larger operand.

Most of this type conversion is performed by the compiler, although in Chapter 8 we shall present memory operations that perform some of these conversions.

6.2 Summary of C Operators

The C operators consist of several basic groups which we address in separate sections of this chapter.

- Bitwise
- Arithmetic
- Relational
- Logical

Arithmetic Operators

The arithmetic operators are:

+ Addition

- Subtraction

/ Division

* Multiplication

% Modulo

Neither of the division operations (/ and %) are implemented directly in the Cortex-M0 instructions set, but are provided in `libgcc` by the compiler.

C Relational Operators

The relational operators are:

== Equal to.

!= Not equal to.

> Greater than.

< Less than.

>= Greater than or equal to.

<= Less than or equal to.

C Logical Operators

The C logical operators are somewhat unusual – the AND and OR are defined to execute sequentially which means that whether or not the second operand is evaluated depends upon the result of evaluating the first operand. It is conveent to implement each logic operation in terms of previously discussed operations.

C expression	Description	Equivalent Behavior
A && B	Logical AND	A ? B : 0
A || B	Logical OR	A ? 1 : B
!A	Logical NOT	A == 0

6.3 Bitwise Logic Operations

The C language defines four bitwise logic operators.

`&` AND

`|` Inclusive or

`^` Exclusive or

`~` Bitwise NOT (one's complement.

There are five bitwise logical operations in the Cortex-M0 instruction set – and, or , bit clear, exclusive or, and "move negative" (bitwise negation) that operate on words (signed or unsigned integers). The syntax for these is:

```
ands rn, rm
orrs rn, rm
eors rn, rm
bics rn, rm
mvns rn, rm
```

We present all of these instructions with the "s" suffix – this is required for the thumb unified syntax, but was not required in earlier versions of the thumb syntax.

`rn` is the register holding the first operand and also the destination register.

`rm` is the second operand register

These instructions reference only registers 0-7. We can capture the (basic) behavior of these instructions in C as follows.

```
rn = rn & rm;  // ands rn, rm
rn = rn | rm;  // orrs rn, rm
rn = rn ^ rm;  // eors rn, rm
rn = rn & ~rm; // bics rn, rm
rn = ~rm;      // mvns rn, rm
```

All of these modify the N and Z condition flags in the APSR register as a side effect. Examples 6.1 and 6.2 illustrate the behavior of two logical instructions `eors` and `bics`. The examples include the state of the affected registers both before (pre) and after (post) execution. Notice that the status bits are lower-case for 0 and upper-case for 1. In each example, `r0` is the destination register for an operation involving both `r0` and `r1`. In Example 6.1, instruction execution affects the `N` (negative) status bit.

We can test these instructions with assembly level procedures using the qemu-arm simulator. Figure 6.1 illustrates a simple program and C test

Example 6.1: Exclusive Or

```
pre   flags = nzcv
        r0  = 0x00000007
        r1  = 0x80000001

      eors r0, r1

post  flags = Nzcv
        r0  = 0x80000006
        r1  = 0x80000001
```

Example 6.2: Bit Clear

```
pre   flags = nzcv
        r0  = 0x00000007
        r1  = 0x80000001

      bics r0, r1

post  flags = nzcv
        r0  = 0x00000006
        r1  = 0x80000001
```

harness to experiment with the `orrs` instruction. There are a number of new concepts embedded in this example; however, you may ignore most of them for now. The key ideas are that the assembly routine declares a global name:

```
        .global or
or:
```

uses registers `r0` and `r1` as operands – the result of the `orrs` instruction is returned in `r0`.

```
        orrs r0, r1
```

Procedure return is handled by moving the return address stored in `lr` to the program counter `pc`. We will discuss this in greater detail in subsequent chapters – for now you should treat this as a simple pattern.

```
        mov  pc, lr
```

```
  .text
  .code 16
  .syntax unified
  .align 2
  .global or
or:
  nop
  orrs r0, r1
  mov  pc,lr
  .end
```

```
#include <stdio.h>
extern int or(int, int);
int data[] = { 0x01, 0x04, 0x05, 0x0A };

void main() {
  int i = 0;
  while (i < 4) {
    printf("%x OR %x = %x\n", data[i], data[i+1], or(data[i],data[i+1]));
    i += 2;
  }
}
```

The result of compiling and running the test program is:

```
/l/arm/gcc-arm-embedded/bin/arm-none-eabi-gcc -g -mcpu=cortex-m0 -mthumb
    ↪-o c-or.elf main.c or.s -specs=rdimon.specs -lc -lrdimon
qemu-system-arm -cpu cortex-m3  -semihosting -kernel c-or.elf
1 OR 4 = 5
5 OR a = f
```

Figure 6.1: Testing Logical Instructions

Notice that the C test routine calls our assembly routine with two operands – the compiler causes these to be placed in `r0` and `r1` – and expects a return value, which the compiler expects to be in `r0`. To keep the compiler from issuing warnings about undeclared procedures, our C test routine includes an extern declaration:

```
extern int or(int, int);
```

It is easy to adapt this example to test other instructions by modifying the C and assembly declarations, the key assembly instruction (`orrs`), and writing new test cases in C.

Exercise 6.1: Bitwise Logical Operations

Write test programs for each of the other bitwise logical operations. Make sure to test a few interesting cases. Show, by expanding the low order bytes as bit strings, that the results make sense for at least one case for each instruction.

6.4 Move Instructions

There are a small number of Cortex-m0 instructions for moving constants into registers and values between registers. We can load small positive constants (0..255) into `r0..r7` with:

```
movs rd, imm8
```

Notice that the `N` and `Z` flags will be set based upon the value of the immediate.

Values can be moved between registers `r0..r7` with setting flags:

```
movs rd, rm
```

or between a "high" register (`r8..r15`) and a "low" register without setting flags:

```
mov rd, rm
```

6.5 Shift Instructions

There are three Cortex-M0 shift instructions and one *rotate* instruction (that we do not consider here). The shift instructions are `lsls` (logical shift left), `lsrs` (logical shift right), and `asrs` (arithmetic shift right). `lsls` and `lsrs` are equivalent to C left and right shift operations on unsigned integers and `asrs` is equivalent to the C right shift on a signed quantity.

All of the shift operations affect the C, N, and Z status flags. The carry flag C is defined by the last bit shifted out, the N and Z flags are defined by the result. Consider the example

```
lsrs r0, r1, 3
```

The meaning of this operation in C is

```
  C = (r1 >> 2) & 1;
r0 = (unsigned int ) r1 >> 3;
```

Thus unsigned integer `r1` is divided by 2^3 – the high order bits are replaced by 0's.

The arithmetic shift right instruction differs in that the high order bits are replaced by copies of the sign bit. Thus,

```
asrs r0, r1, 3
```

is defined as

```
 C = (r1 >> 2) & 1;
r0 = (int) r1 >> 3;
```

The logical shift left instruction is used with both signed and unsigned operands:

```
lsls r0, r1, 3
```

is defined as

```
 C = (r1 >> 29) & 1;
r0 = (int) r1 << 3;
```

The carry bit is the last bit shifted out; low order bits are replaced by zeros.

These operations are all available with both integer and register operands:

```
lsls rd, rm, imm   @  rd = rm << imm
lsls rd, rm        @  rd = rd << rm
lsrs rd, rm, imm   @  rd = (unsigned) rm >> imm
lsrs rd, rm        @  rd = (unsigned) rd >> rm
asrs rd, rm, imm   @  rd = (signed) rm >> imm
asrs rd, rm        @  rd = (signed) rd >> rm
```

In all cases, the behavior is defined for shift amounts in the range 1 to 31. For right shift operations, shift amount 0 is interpreted as sift by 32 (hence the carry bit is shifted in). Where the shift amount is determined by a register, only the lower byte of the register is used to define the shift.

The formats of the `lsl` instructions are illustrated in Figure 6.2. The formats of the other shift and rotate instructions differ only in the decoding bits.

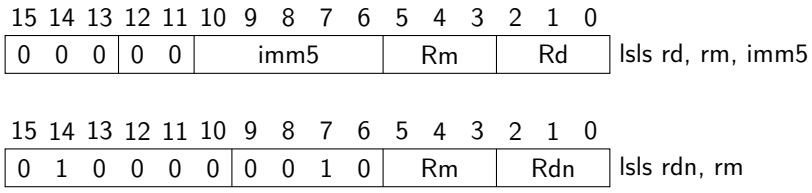


Figure 6.2: Format of Left-shift Instructions

Exercise 6.2: 64-bit Shift Operations

Write a C program that shifts a 64-bit integer operand by a programmable amount using only shifts and C bitwise operators on 32-bit operands.

```
uint64_t asl(uint64_t op, unsigned char amount);
```

Hint, you can access the halves of a 64-bit integer using a union.

```
#include <stdint.h>
typedef union {
     int64_t op;
     int32_t halves[2];
} operand;
```

6.6 Arithmetic Operations

C defines the following arithmetic operations – addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). Neither division nor modulus are directly supported by the Cortex-M0 instruction set, but are provided by the compiler through `libgcc`.

While the encoding of the logical operations is easy to understand – there is only one possible format – the encoding of addition and subtraction is relatively complicated because there are multiple encodings for various combinations of operands. Where the logical operations were limited to two register

Exercise 6.3: Division

Write a C program to perform 32-bit integer division using only addition, subtraction, and shifting operators.

operands chosen from r0-r7, there are versions of addition and subtract for two and three operands, and operands other than r0-r7 including sp, pc, and constants. Constants are signed or unsigned integers encoded in a small number of bits – all of the arithmetic constants are unsigned.

Rather than examining the encoding of all these variations, the following table summarizes the key points for addition. Notice there are various limits on constants – imm3 is a 3-bit unsigned number, imm7 is a 7-bit unsigned number, imm8 is an 8-bit unsigned number. The exact number of bits and their interpretation is dependent upon the instruction format – the sixteen instruction bits must be shared by the format identification bits and all of the various operands. [1]

adds Rd, Rn, imm3	Rd = Rn + imm3
adds Rd, imm8	Rd = Rd + imm8
adds Rd, Rn	Rd = Rd + Rn
add Rd, Rn	Rd = Rd + Rn [2]
adds Rd, Rn, Rm	Rd = Rn + Rm
add sp, imm7	sp = sp + (imm7«2)
add Rn, sp, imm8	Rn = sp + (imm8«2)
adcs Rd, Rm	Rd = Rd + Rm + Carry

`adcs` is an important instruction for multiprecision arithetic because it adds two numbers with the result of a previous operation. Here are a few examples of the various add instructions:

```
.syntax unified
 adds r0, r1, 3
 adds r0, r1, r2
 adds r0, r1
 adds r0, r1
 add r10, r11
 add pc, r10
 adds r0, 222
 add sp, 64
 adcs r1, r2
```

[1] The architects of the ARM and Thumb instruction sets had to weigh the importance of constants against the ability to encode a rich instruction set. It's clear that some constants (.e.g. -1,0,1,2,4 ...) are more frequently used that others (e.g. 1033). It is possible to access arbitrary constants, just not with these instructions. In the general case, an arbitrary constant must be loaded from memory into a register, and the operation performed with that register.

[2] One of Rd, Rn must be a high register – r8-r15.

Notice that it is possible to perform addition on a pair of "high" registers such as r10, r11. Further, while it is possible to add a register to the program counter (pc), this is rarely advisable. There is a special case of computing the address of a location that is a constant offset from the program counter that we will consider in a subsequent chapter.

Also notice that there are two forms of addition `adds` and `add` – the former updates the status flags, while the later does not. While the full ARM instruction set supports both forms in all cases, the Cortex-M0 instructions are far more restrictive. The various ARM documents are unclear about which of the add instructions modify the flags. It's easiest to enumerate the ones that don't. Operations that use one of the high registers (any register other than r0-r7) do not modify the flags. With unified syntax enabled,

the GNU assembler enforces legal instruction forms; "s" means the flags are modified, no "s" means the flags are not modified. Thus

```
.syntax unified
     ...
     adds r10, r11
```

will cause an assembler error because the Cortex-M0 only supports addition of "high registers" without setting flags.

The use of constants is likely to be confusing to a novice programmer. Notice the special case of adding a value to `sp`. In this case the constant (immediate) value (`imm7` or `imm8`) is multiplied by 4 – left shifted by 2. All data on the stack is assumed to be word aligned, so there is no point in supporting the addition of constants that are not divisible by 4. For adding to `sp`, constants 0-1024 that are divisible by 4 are possible.

In general, the only way for a programmer to understand the limits of constants is to examine the instruction formats. For example, the two "add immediate" instruction formats are illustrated in Figure 6.3.

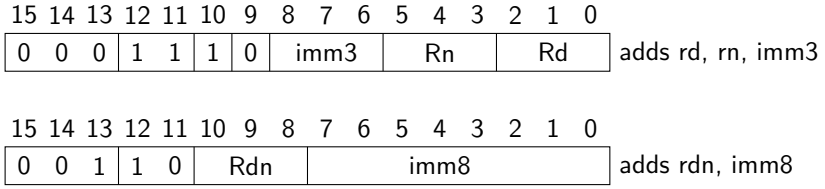


Figure 6.3: Two of the add immediate formats

The corresponding subtraction operations are shown in the following table. The `rsbs` instruction stands for "reverse subtraction"; the only legal form for the Cortex-M0 is as a negation operator, but the general ARM instruction allows other combinations of operands. The form given, with the apparently redundant 0, is to ensure that the same assembly instruction is legal for all ARM processors.

subs Rd, Rn, imm3	Rd = Rn - imm3
subs Rd, imm8	Rd = Rd - imm8
subs Rd, Rn	Rd = Rd - Rn
subs Rd, Rn, Rm	Rd = Rn - Rm
sub Rn, sp, imm8	Rn = sp - (imm8«2)
sbcs Rd, Rm	Rd = Rd - Rm - Carry
rsbs Rd, Rn, 0	Rd = -Rn

Examples include:

```
.syntax unified
    subs r1, r2, 6
    subs r1,  127
    subs r0, r1, r2
    subs r0, r1       @ same as subs r0,r0,r1
    sub sp, 44
    sbcs  r1, r2
    rsbs r0, r1, 0    @ 0 is required
```

`C` and `V` Flags

The effect of the addition and subtraction operations on the carry and overflow flags can be confusing (especially the latter). Consider the case of addition. When adding two 32-bit numbers, the carry flag is set if there is a carry-out from adding bits 31. The overflow flag is interpreted in the context of adding two signed numbers – if two numbers of the same sign (bit 31 is equal) are added and the result has a different sign, then the overflow flag is set. For signed arithmetic, overflow means that the result was too large to represent in 32-bits. For unsigned arithmetic, carry out means that the result was too large to represent.

Subraction, `(A-B)` is interpreted as addition with the two's complement of the second operand – `(A + ~B + 1)` – and the flags are then interpreted as for addition. Subtraction with carry `sbc` replaces the 1 above with the carry bit `(A + ~B + C)`.

Work through some examples showing how the flags are set

Write a C program that multiplies 64-bit integers by a series of 32-bit multiplications and 64-bit additions.

The Cortex-M0 provides a 32-bit multiplication instruction:

```
muls rd, rn    @  rd = rd * rn
```

Chapter 7

Conditional Execution

In this chapter we introduce the instructions necessary to support conditional execution; e.g. `if` and `while` statements in C. In the proceeding chapter we discussed the data processing instructions and described how they affect the condition flags (N,V,C,Z). In this chapter we show that a single additional instruction, in conjunction with these condition flags, can be used to implement all of the C if statements and looping constructs. This may seem somewhat surprising; we begin by showing that all these can be realized within C.

While most readers will be familiar with the C control-flow operations such as `if-else`, `while`, `until`, and `for`; C has another, frequently disparaged, control-flow statement – `goto` which has the form:

```
  goto label:
..

label:  statement;
```

The `label` can be anywhere within the same procedure and may be any text that is not a C keyword.

Suppose that we wish to find the minimum of two numbers `x` and `y`. We could write this in C as

```
if (x > y)
    min = y;
else
    min = x;
```

We could rewrite this as

```
      if (x > y) goto THEN;
         min = x;
         goto END;
THEN: min = y;
END:
```

Because branches are expensive, it is actually better to do this:

```
      min = y;
      if (x > y) goto END;
          min = x;
END:
```

The Cortex-M0 instruction set provides a conditional branch instruction that enables the implementation of such operations. The branch instruction has the form:

```
b<c>  <label>   @  if (c) goto <label>
```

Where `c` is a condition (the empty condition means "always") and the label is a program location following the current instruction. For example, we can implement the `min` code as follows. Assume that `r0=y` and `r1=x`, and at the end `r0=min`; we can write this code fragment in the Cortex-M0 instruction set as:

```
                        @ min == r0, r0 == y
        ...             @ evaluate condition (postponed)
       bgt .END         @ if (x > y) goto .END
       movs r0, r1      @ min = x
.END:
```

In the preceding example, we deferred discussion of the *branch condition*. Briefly, the condition is computed from the status flags set by preceding instructions. For example,

```
       subs r2, r1, r0   @ r2 = x - y
       bgt .END          @ if (x > y) goto .END
       movs r0, r1       @ min = x
.END:
```

In the preceeding example, `r2` was used to store a result that is not needed (only the flag values are used). There are three instructions that are provided to evaluate expressions where only the side effects are preserved – `cmp` and `cmn`, which perform comparisons, and `tst` which performs a bit test. `cmp` is equivalent to subtract and `cmn` to addition, but without storing the result. The primary function of `tst` is to test bits in an arbitrary binary format – for example, decoding instruction formats.

cmn Rn, Rm	Rn + Rm
cmp Rn, Rm	Rn - Rm
cmp Rn, imm8	Rn - imm8
tst Rn, Rm	Rn & Rm

Thus we can rewrite our example to:

```
        cmp r1, r0          @ x - y
        bgt .END            @ if (x > y) goto .END
        movs r0, r1         @ min = x
.END:
```

The actual flag values corresponding to `>` are complicated to derive. It's clear that the `Z` flag should be zero, but what about `C,N,V` ?

In the remainder of this chapter, we discuss the computation of conditions and show how these can be used to implement the C logical and relational operators. We then consider the various C control flow operations in turn.

7.1 Condition Codes

The Cortex-M0 defines 16 *condition codes* as combinations of status flag states. All of these are determined from subtracting one operand from another. In some cases, there meaning is dependent upon whether the operands represent signed or unsigned numbers.

Suffix	Flags	Meaning
eq	Z = 1	Equal, last result was zero
ne	Z = 0	Not equal, last result was not zero
cs or hs	C = 1	Higher or same, unsigned
cc or lo	C = 0	Lower, unsigned
mi	N = 1	Negative
pl	N = 0	Plus or zero
vs	V = 1	Overflow
vc	V = 0	No overflow
hi	C = 1 and Z = 0	Higher, unsigned
ls	C = 0 or Z = 1	Lower or same, unsigned
ge	N = V	Greater than or equal, signed
lt	N != V	Less than, signed
gt	Z = 0 and N = V	Greater than, signed
le	Z = 1 and N != V	Less than or equal, signed
al	-	Always, this is the default when no suffix is specified

Exercise 7.1: Condition Flags

Using the definitions for 2's complement and the status flags, prove that the condition codes eq, ge, and gt are properly defined.

7.2 C Relational Operations

Recall that in C, the logical value of an expression is True if the value of the expression is non-zero and False otherwise. For arithmetic expressions, the logical value is captured by the Z status flag as a side effect of expression evaluation. For example, we might decrement a counter and test if the result is zero as:

```
next: ...
      if (--counter) goto next;
```

Which can be realized in assembly as

```
next:
       subs r0, 1
       bne next
```

C defines a complete set of relational operators

== Equal to.

!= Not equal to.

> Greater than.

< Less than.

>= Greater than or equal to.

<= Less than or equal to.

It is not surprising that all of these can be realized in assembly as variations of conditional branches. Returning to the previous example, but changing the condition:

```
next: ...
      if (--counter >= 0) goto next;
```

Which can be realized in assembly as

```
next:
      subs r0, 1
      bge next
```

More generally, we might want to compare two expressions. For example `x <= y`. We do this by computing the two operands separately:

```
        @   r0 = x
        @   r1 = y
        movs r2, 0          @ default result is 0
        cmps r0, r1         @ x - y
        blt  skip           @ x < y ? skip next statement
        movs r2, 1          @ r2 = 1
    skip:
```

Here we have stored the result of the relational operation in `r2`, which is necessary if the flag values aren't used immediately.

In summary, we compute the relational operators either as a side effect of an arithmetic operation (when comparing with 0), or by an explicit comparison of the two operands of the relation. We can used the result immediately in a conditional branch – possibly using the branch to store the result required by C (a 0 or 1).

C Logical Operations

The logical AND and OR operators (&&, ||) are somewhat unusual in requiring sequential execution – whether or not the right operand is evaluated depends upon the value of the left operand. For example,

```
if (a && b)
   stmt
```

is implemented as

```
if (a != 0)
   if (b != 0)
       stmt
```

The replacement of operands `a` and `b` with comparisons is consistent with the C semantics. Both logical operators can be realized by such comparisons.

7.3 C Control Flow

We now have all the assembly language components that we need to implement the various C control flow operations. In the following we examine

a variety of these C operations; in each case, we first rewrite the C code to utilize `goto` statements. The translation to assembly language is relatively straightforward.

Consider the general `if-else` statement:

```
if (<cond>) {
   <then-body>
} else {
   <else-body>
}
<after>
```

Where text of the form `<name>` is intended as a place-holder for an arbitrary expression (for conditions) or sequence of statements (for code). We can translate this into the following equivalent program fragment:

```
   if (<cond>) goto THEN:
      <else-body>
   goto L1;
THEN:
      <then-body>
L1:
   <after>
```

For the simple case with no `else` it may be desirable to use the following form:

```
   if (!<cond>) goto L1:
   <then-body>
L1:
   <after>
```

As we have seen, evaluation of an arbitrary expression may take multiple simpler statements. In general, we assume that this can be handled as follows:

```
   c = cond;
   if (!c) goto L1:
   <then-body>
L1:
   <after>
```

While Loops

The simplest C-loop form is the `while` loop. A generic loop has the form:

```
while (<cond>) {
  <while-body>
}
```

This is easily translated into a conditional statement and goto:

```
L1:
  if ( <cond> ) {
    <while-body>
    goto L1;
  }
```

For example,

```
  while (count <= 10) {
     count++;
   }

...

L1:
  if ( count <= 10 ) {
    count++
    goto L1;
  }
```

In the case of a complex expression, we might implement it as:

```
L1:
  c = <cond>
  if ( c ) {
    <while-body>
    goto L1;
  }
```

One small issue with this naive translation is that that, when executed, every loop iteration has to execute two "branch" statements:

```
L1:
  if ( <cond> ) {
    <while-body>
    goto L1;
  }

...

L1:
  if (!<cond>) goto L2
     <while-body>
  goto L1:
L2:
```

We can get improved performance by a simple rearrangement:

Exercise 7.2: Translating `do-while`

In contrast with the C `while` statement, which executes as long as the specified condition is true at the beginning of the loop, the C `do while` statement executes as long as the condition is true at the end of the loop. Show how to translate the `do while` statement into conditional plus goto form.

```
  if (!<cond>) goto L2
L1:
    <while-body>
  if (<cond>) goto L1
L2:
```

While this may result in duplicated code (for expression evaluation), this is a "static" cost as opposed to the "dynamic" cost imposed by executing multiple goto instances per loop iteration. In the subsequent discussion, we will ignore such optimization issues.

For Loops

The other commonly used loop structure in C is the for loop which has the form

```
for ( <init> ; <cond> ; <update> ) {
  < for-body>
}
```

Which is equivalent to:

```
  <init>
  if (!<cond>) goto EXIT
L1:
  <for-body>
UPDATE:
  <update>
  if (<cond>) goto L1;
}
EXIT:
```

`continue` and `break` are implemented as simple `goto` statements.

```
  goto EXIT;   // break

  goto UPDATE; // continue
```

Work through basic C control flow examples.

7.4 Switch Statements

The C Switch statement is generally implemented in two ways – either as a series of conditional statements, or as look-up table. In the later case, a table entry is made for each possible value to the switch condition containing the address of code to jump to. Since we have not yet discussed memory operations (Chapter 8), we focus on the more common case here.

Consider the simple switch statement:

```
switch(x) {
   case 1 :  x = 2; break
   case 2 :  x = 3; break
 default:    x = 4
 }
```

```
  cmps r0, 1      @ x == 1
  bne  c2
  movs r0, 2
  b    end
c2:
  cmps r0, 2
  bne  default
  movs r0, 3
  b    end
default:
  movs r0, 4
end:
```

7.5 Procedure Calls

Procedure calls in the Cortex-M0 are implemented with a "branch and link" instruction. This is similar to the branch instruction presented previously, but, as a side-effect, it saves the address of the following instruction in the link register (`LR`).

```
bl label      @  pc = label, lr = next inst.
```

In previous examples we've used the `bx` (branch exchange) instruction to return from procedures without explanation. Its function is to move the contents of a register (in this case the link register) to the program counter. It can also be used for unconditional branches to locations beyond the range of the regular branch instruction.

```
bx  lr        @ pc = lr
```

7.6 Branch Instruction Encoding

The encoding of branch instructions is somewhat more complicated than the data processing instructions. Consider, for example, a basic conditional branch instruction:

```
    bgt label
    ...
label:
```

The obvious question is how is the `label` stored in the instruction word ? The answer is somewhat surprising – the actual label is not stored directly, but rather an offset, from the program counter of the instruction, is stored instead. The address of the label is computed by the instruction interpreter. The name for this form of addressing is "PC Relative" meaning an address is computed from an offset and the current program counter.

There are two key advantages to this approach. First, the size of the instruction field can be much smaller (at the expense of range). Second, the code is position independent – the linker can assign the code to any address without changing its behavior. For conditional branches, this offset is stored as an 8-bit signed number premitting offsets from the PC of -256 to 254 (even numbers only !). For unconditional branches, the range is -2048 to 2046. The corresponding instruction formats are shown in Figure 7.1.

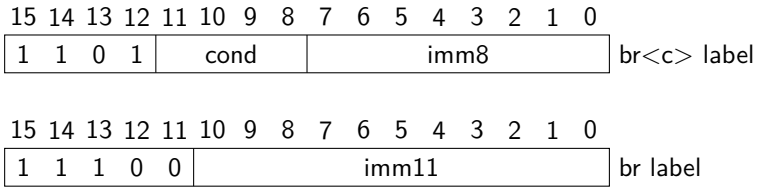


Figure 7.1: Branch Instruction Formats

Branch and link is a rare 32-bit Cortex instruction. This instruction can encode offsets in the range -16,777,216 to 16,777,214. The actual offset is computed (and set) by the linker rather than the assembler. Because the offset distance is not known until link time and it is possible that the necessary offset is out of range, it is possible for the linker to generate special "trampoline" code using a reserved (Hi) register to handle this rare event.

Chapter 8

Memory Reference

In the preceding chapters, we have focused upon small examples where all the required data were provided by a C test harness and ignored the issues related to accessing data stored in memory. In this chapter we investigate the machine *memory reference* instructions that provide the ability to read and write data from memory.

The fundamental abstraction is pointer de-referencing – given the address of a memory location, we can read or write words (or smaller) from that location. For example:

```
extern int *p;
int tmp;
*p = exp;        /* write a pointer */
tmp = *p + 3;    /* read  a pointer */
```

As we shall see, variable name in a C program is replaced by the compiler by a reference to the memory location providing the storage for that variable.

In the C language, data are allocated either statically by the compiler (and linker), or dynamically at runtime. Static data are those declared outside of procedures, or within a procedure, but with the `static` keyword. Dynamic data are declared within procedure bodies (without the `static`), and hence allocated at procedure entry; an additional category of dynamic data, storage allocated on the heap (e.g. by `malloc`), is out side of the scope of this chapter.

In this chapter we primarily focus upon static data – the C runtime stack, which provides the storage for dynamic data, is discussed in detail in the next chapter. A typical C program consists of a number of modules (files) that are compiled individually and then linked together into a single executable. For example, our test programs have consisted of a C module

(the test harness) and a assembly module to be tested. It is the job of the compiler to reserve space for static variables declared within each module and to create global symbols for those variables with global scope. The linker then allocates specific memory locations to satisfy the reservations made by the compiler, and "patches" any cross-module references. As we shall see, there are a number of details that must be handled in order to reduce this model to practice.

This chapter is organized as a series of small steps – each of which resolves another layer in the static data story. For example, we initially assume that the address for a memory location is provided in a register and only later consider the question of how the compiler/linker provides such an address to the executing program. We first consider how to access integers, then fields within a structure, and finally bytes and half-words. After pointers provided in a register, we consider pointers stored in the program that refer to data allocated in other modules, and then discuss how to allocate data within a module.

8.1 Accessing Words in Memory

We begin our study of static variables with integer pointers. To postpone the discussion of where these pointers come from, we initially consider the case of a procedure which accepts a pointer parameter. Consider Example 8.1. This example consists of a procedure that accepts a single pointer parameter (`p`); it increments the integer *pointed to* by `p`, and returns this result. Accessing a variable via a pointer is called *pointer dereferencing*.

Notice that the assembly program has no types – it loads a word from memory, increments the word (the same program will work for both signed and unsigned integers), and writes the resulting word back to memory.

Example 8.1: Pointer Dereferencing

```
int incptr(int *p) {
        *p = *p+1;
        return *p;
}
```

```
incptr:
     ldr  r3, [r0]  @ r3 = *r0
     adds r3, 1     @ r3++
     str  r3, [r0]  @ *r0 = r3
     movs r0, r3    @ r0 = r3
     bx   lr        @ return
```

8.2 Access an Array Element

In C, we commonly want to reference an address that is relative to a pointer – that is offset by a fixed amount. For example, we may wish to access the 4th element of an array as in Example 8.2. The general form of the load and store instructions discussed in the previous example is:

```
ldr rt, [rn,offset]    @ rt = *(rn + offset)
str rt, [rn,offset]    @ *(rn + offset) = rt
```

However, for word load and store, the offset must be a multiple of 4 – the constant stored in the instruction is `offset >> 2`.

Example 8.2: Array Access – Constant Offset

```
int elcopy(int p[]) {
        p[3] = p[4];
        return p[4];
}
```

```
elcopy:
   ldr r3, [r0,16]   @ r3 = *(r0 + 16)
   str r3, [r0,12]   @ *(r0 + 12) = r3
   mov r0, r3        @ r0 = r3
   bx lr             @ return
```

Example 8.3: Array Access – Variable Offset

```
void elswap(int p[], int el) {
    int tmp = p[el];
    p[el]   = p[el+1];
    p[el+1] = tmp;
}
```

```
elswap:
    lsls r1, 2        @ r1 = el * 4
    adds r3, r0, r1   @ r3 = &p[el]
    ldr  r2, [r3]     @ tmp = p[el]
    adds r1, 4        @ r1 = el + 4
    adds r0, r1       @ r0 = &p[el+1]
    ldr  r1, [r0]     @ r1 = p[el+1]
    str  r1, [r3]     @ p[el] = p[el+1]
    str  r2, [r0]     @ p[el+1] = tmp
    bx   lr
```

In the general case, we may wish to access an array element indexed by a variable. Consider Example 8.3, in which two array elements are swapped. In this example, many of the instructions are focused upon pointer arithmetic. Notice that the first instruction multiplies `el` by four using a shift operation – recall that C arrays are indexed by element number, while memory is addressed by byte. Thus the memory offset must be computed by multiplying the array index by the element size. Suppose that we wished to index an array of 16-bit elements, in this case the index would be multiplied by two. Similarly, for 64-bit elements, the index would be multiplied by eight.

The Cortex-M0 is somewhat impoverished with respect to *addressing modes*, i.e. the instruction formats that simplify address calculations. The general ARM instruction set provides memory reference instructions that compute addresses by adding two registers, one of which is shifted (multiplied) by an appropriate constant.

8.3 Accessing Fields in a Structure

Another common memory access pattern occurs with structures. In Chapter 2 we described how a structure is stored as a block of memory, with each field a known, fixed offset from the beginning of the block. Fields in a structure are accessed using normal memory reference instructions with a

Example 8.4: Accessing Structure Fields

```
typedef struct {int x, y;} Point;
typedef struct {Point ur,ll;}  Rect;

int perimeter(Rect *r) {
      return 2*(r->ur.x - r->ll.x) +
             2*(r->ll.y - r->ur.y);
}
```

```
perimeter:
  ldr  r3, [r0]     @ r3 = r->ur.x
  ldr  r2, [r0, 8]  @ r2 = r->ll.x
  subs r2, r3, r2   @ r2 = r3 - r2
  ldr  r3, [r0, 12] @ r3 = r->ll.y
  ldr  r0, [r0, 4]  @ r0 = r->ur.y
  subs r0, r3, r0   @ r0 = r3 - r0
  adds r0, r2, r0   @ r0 = r0 + r2
  lsls r0, r0, 1    @ r0 = r0 * 2
  bx   lr
```

pointer plus offset. Consider Example 8.4 that computes the parameter of a rectangle. The rectangle consists of two points each of which consists of `x` and `y` coordinates. As is common for graphics, the point (0,0) is in the upper-left corner.

Exercise 8.1: Structure Access

The objective of this exercise is to create a Set data structure using linked lists. Given the following list structure:

```
typedef struct CELL *SET;
struct CELL {
       int element;
       SET next;
};
```

Part 1 Implement iterative functions for the following operations in C.

```
int lookup(int x,  SET S);   // return true if x is in S
CELL insert(SET X, SET *pS); // insert X in pS
                             // Return X if it would
                             // duplicate an element in
                             // the set and 0 otherwise.
CELL delete(int x, SET *pS); // delete x from pS and return
                             // the containing CELL (if any)
```

The slightly unusual prototypes are designed so that all allocation can occur in the test wrapper. For example, to insert a value, the test wrapper will create a cell with that value and attempt to insert it into the set. The insert function returns either 0, if the value is unique, or the cell if not. Both can be safely passed to `free`.

Part 2 Implement your C functions to assembly and test. It's best if you replace one function at a time and test. You will probably need to used GDB to complete this assignment.

Part 3 Convert your C functions and their assembly implementations to recursive functions – you'll need to read the beginning of the Chapter 9 to see how to save and restore `lr`.

8.4 Loading Addresses and Constants

In the preceding sections, we have assumed that the addresses necessary to access statically allocated objects in memory were available. In this section, we demonstrate how these addresses can be loaded from memory into a register by a program. As we shall see, the same technique is used to load arbitrary constants (i.e. constants that are too large to be directly encoded in and instruction word). Consider Example 8.4. The key new instruction is:

Loading Addresses

```
extern int x;

void xinc(void) {
    x = x + 1;
}
```

```
xinc:
     ldr  r2, .L2   @ r2 = &x
     ldr  r3, [r2]  @ r3 = x
     adds  r3, 1    @ r3++
     str  r3, [r2]  @ x = r3
     bx   lr        @ return
    .align  2       @ bump address
.L2:                @ local label
     .word   x      @ &x
```

```
ldr rt, label
```

which is actually an assembler shortcut for

```
ldr rt, [pc,offset]
```

This instruction loads a word from the address at `pc + offset` where `offset` is a positive number, divisible by four, in the range 0-1024. This addressing *mode* is called *pc relative* because it applies a relative offset to the current program counter to compute the address. The pointer we wish to load is stored (by the assembler/linker) at the location associated with the `label`. The advantage to the `label` form is that the assembler calculates the correct `offset` value, which can be quite challenging to correctly compute by hand in many cases.

Returning to the example;

```
    .align 2
.L2:
     .word  x
```

The `.align 2` is an assembler directive that forces the following code/data to be aligned on a 4-byte (2^2) boundary. `.L2` is a label – it is common, but not required, that local labels start with ".". The final directive inserts a word of data in the current (code) segment, at the location associated with label`.L2`. The inserted data is `x`, the address of the program variable by the same name.

Loading Constants

```
int inc(int x) {
    return x + 1000000;
}
```

```
inc:
  ldr r3, .L2
  add r0, r3
  bx  lr
        .align  2
.L2:
  .word 1000000
```

Note that the compiler translates the *names* of objects – procedures or variables – into addresses. At the assembly level, the name `x` is represented by an address. When this assembly routine is translated into object code, it will be the job of the assembler to insert the correct address of `x` at the location marked by `.L2`.

In summary, the basic approach to loading a static address is to insert in the code segment, immediately following a procedure that requires that address, a label and data word containing the address. The address is then loaded into the running program with a *pc relative* load instruction. The assembler greatly simplifies our lives by performing the key offset calculation; the linker ensures that all symbolic references (e.g. a variable name) are correctly translated into physical addresses.

The same technique is used to load large constants (those that don't fit in the instruction word). This is illustrated in Example 8.4.

8.5 Allocating Storage

In the preceding section, we showed how to load the address for a variable from the code segment. The only remaining issue is to allocate space for a variable. The assemble supports static allocation through the directive `.comm` which declares a *common symbol*. [1] A common symbol in one object file may be linked to the same name in another object file. The syntax in the gnu assembler is:

[1] This terminology dates to the early fortran era.

```
    .comm name, size, alignment
```

This allocates a block of memory of the declared size (in bytes) in the BSS section and alignment (in bytes) with the (global) symbol name. Returning to Example 8.4, we can modify the assembly code to allocate the storage for `x`.

```
            .align  2
    .L2:
            .word   x           @ address of x
            .comm   x,4,4       @ allocate 4 bytes,
                                @ aligned 4,
                                @ with name x
```

Note that the pointer to `x` and the storage for `x` will not be in the same memory region – the pointer will be linked into the code section (immediately following the preceding procedure), will common blocks are allocated space in the data section.

The situation is somewhat different for static variables; although the method for loading the address into a program is the same.

```
        .align  2
  .L2:
        .word   x               @ store x pointer
        .bss                    @ switch to BSS section
        .align  2               @ force alignment
    x:
        .space  4               @ allocate four bytes
```

The directive `.space 4` allocates four bytes.

If we wished to both allocate `x` and initialize it (e.g. to 3), we need to allocate space in the data section

```
        .align  2
  .L2:
        .word   x           @ store x pointer
        .data               @ switch to DATA section
        .align  2           @ force alignment
    x:
        .word   3           @ allocate word with value 4
```

To create a global variable that is initialized, we need only add a directive declaring `x` to be a global symbol (i.e. visible to the linker).

```
        .align  2
  .L2:
```

```
    .word   x          @ store x pointer
    .global x          @ declare x global
    .data              @ switch to DATA section
    .align  2          @ force alignment
x:
    .word   3          @ allocate word with value 4
```

8.6 Accessing Half-words and Bytes

We have been restricting our focus to word-sized memory accesses; in this section we present instructions for accessing half-words and bytes. As you have probably realized, there are essentially no data processing operations for anything other than words (with the exception of sign extension and byte-swap operations), and the C language dictates that shorts and chars are converted to integers (signed or unsigned) before any computation. However, it is necessary to be able to load and store these quantities.

The Cortex-M0 instruction set provides support for loading and converting both signed and unsigned quantities in a single operation. For signed quantities, this requires copying the sign bit to the additional target bytes; for unsigned quantities, this means writing 0 to the additional target bytes.

Syntax	Semantics
`ldrb rd, [rn, imm]`	`rd = *((unsigned char *) rn + imm);`
`ldrb rd, [rn, rm]`	`rd = *((unsigned char *) rn + rm);`
`ldrsb rd, [rn, rm]`	`rd = *((char *) rn + rm);`
`ldrh rd, [rn, imm]`	`rd = *((unsigned short *) rn + imm);`
`ldrh rd, [rn, rm]`	`rd = *((unsigned short *) rn + rm);`
`ldrsh rd, [rn, rm]`	`rd = *((short *) rn + rm);`

Notice the lack of signed loads for immediate offset. There are separate *sign extension* instructions (`sxth, sxtb`) that may be used to "fix" the result of an unsigned load. None of these operations affect the status flags.

The the situation for storing half-words and bytes is similar, but without the need to distinguish signed from unsigned. Each of these instructions truncates its argument – this is the same behavior that C requires.

Syntax	Semantics
`strb rd, [rn, imm]`	`*((unsigned char *) rn + imm) = rd;`
`strb rd, [rn, rm]`	`*((unsigned char *) rn + rm) = rd;`
`strh rd, [rn, imm]`	`*((unsigned short *) rn + imm) = rd;`
`strh rd, [rn, rm]`	`*((unsigned short *) rn + rm) = rd;`

None of these operations affect the status flags.

8.7 Volatile Data

In the preceding sections, operating on static data has required reading the data into memory, modifying those data, and writing them back. If a computation modified some static location multiple times, it would be reasonable for a compiler to optimize the computation in order to minimize the number of writes. Unfortunately, if the data is not memory, but rather a memory-mapped device – for example, a serial output register, this optimization would be completely wrong. With I/O devices, memory operations can potentially have significant side-effects and should not be reordered or optimized. The C `volatile` keyword was introduced specifically for this case.

Chapter 9

Runtime Stack

The core concept of the C runtime is the use of the stack for temporary storage during program execution. In our initial introduction to Cortex-M0 assembly programming, we glossed over this issue. Our examples were restricted to using registers `r0-r3`; we used these registers for passing parameters from a C test harness into simple assembly procedures, all calculations with them, and returned results in `r0`. None of our example assembly routines called other procedures.

In this chapter we show how to use a stack to handle general procedure parameter passing, provide persistent storage between procedure calls, and provide local local storage for procedures. We begin our discussion with a few simple examples, we then introduce a model of the C runtime stack, and show how this model is used to solve the three issues mentioned above. We assume throughout this discussion that the stack is initialized at program entry; initialization consists of allocating a block of storage for the stack and setting the processor stack pointer (`sp`) to the "top" of that memory block. This initialization may be done by an operating system, or, in the case of embedded processors such as the Cortex-M0, by the linker.

9.1 Preserving Registers

We have avoided using registers `r4-r7` for a simple reason – the ARM ABI requires that a procedure that modifies any of these registers must save the register's value on procedure entry and restore the value on return. In the language of compilers, these are *callee saved* registers; i.e. their values are saved (and restored) by the called procedure. In contrast, registers `r0-r3` are *caller saved* registers; a called procedure is free to modify them and may

assume that the calling procedure has saved their values if it needs them in the future.

As an example, consider a simple recursive function that counts the number of 1's in an unsigned integer parameter:

```
int ones(unsigned int i) {
  // if operand is zero, return 0
  if (i) {
    // bit 0 plus recursive call
    return (i & 1) + ones(i >> 1);
  } else {
    return 0;
  }
}
```

Notice that this procedure needs the value of parameter `i` both before and after it makes a (recursive) call to `ones`. An important concept, facilitated by the stack, is that each instantiation of `ones` has its own local storage. Thus, if we call `ones` with the parameter `i == 0x00000003`, that instance of `ones` will recursively call `ones` with the parameter `i == 0x00000001`, but will still have access to the original of `i`.

Recall that (in the simple case) parameters are passed to a procedure in registers `r0-r3`. Thus, in this example, `i` is stored in `r0`. As described above, `r0` is a *caller saved* register. This means that `ones` must save `i` somewhere before making the recursive call. The solution is to use a *callee saved* register (`r4-r7`). The implementation of `ones` moves `i`(`r0`) to `r4` in order to preserve its value; but, it must first save the original value of `r4`.

A similar problem exists for `lr`. Recall that procedure calling is implemented with the *branch and link* instruction (`bl`). This instruction saves a return address (the address of the instruction following the `bl`) instruction in `lr`. However, when `ones` makes a call to `ones`, `lr` will be overwritten! The solution to both these problems is to *preserve* the values of these registers on the stack.

Saving and Restoring Registers

Before returning to our example, we introduce two instructions for accessing memory on the stack. `push` and `pop`. The push operation takes a list of registers to be saved (pushed) onto the stack. This list may include any of the "low" registers (`r0-r7`) as well as the link register (`lr`). The behavior of `push` is illustrated in Example 9.1 which pushes two registers on the stack –

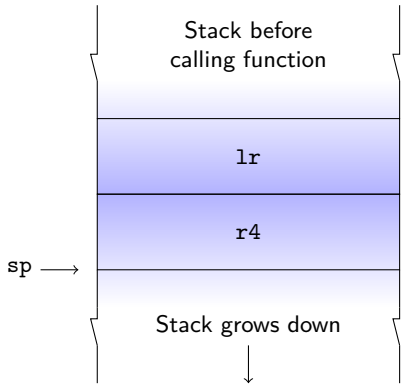


Figure 9.1: Preserving `r4` and `lr`

Example 9.1: Push

```
push {r4,lr}      @ write lr and r4 to the stack
```

PRE r4 = 0x00000003
lr = 0x80000004
sp = 0x00080014

	Address	Data (PRE)	Data (POST)
	0x80018	0x00000001	0x00000001
(pre) →	0x80014	0x00000002	0x00000002
	0x80010	(empty)	0x80000004
(post) →	0x8000C	(empty)	0x00000003

POST r4 = 0x00000003
lr = 0x80000004
sp = 0x0008000c

`r4` and `pc`. These are exactly registers we need to save for our recursive one's count procedure.

Notice that the stack grows downward – from high to low addresses. Because the stack is implemented in memory, no location is every truly "empty"; however, we must assume that any location on the stack below the stack pointer may be overwritten at any time. Hence, in the previous example we showed the two locations below the stack pointer as "empty" prior to executing

Example 9.2: Pop

```
pop {r4,pc}      @ read pc and r4 from the stack
```

PRE r4 = 0x00000007
pc = 0x80000084
sp = 0x0008000C

	Address	Data
	0x80018	0x00000001
(post) →	0x80014	0x00000002
	0x80010	0x80000004
(pre) →	0x8000C	0x00000003

POST r4 = 0x00000003
pc = 0x80000004
sp = 0x00080014

`push`, and showed their contents afterwards.

The dual of the `push` instruction is `pop`, which takes a list of registers to load from the stack. This list may include any of the low registers as well as the `pc`. A common code structure is to push `lr` at the entry to a procedure and to pop `pc` at the end – restoring any saved registers and returning from the procedure in a single operation. Consider the `pop` operation, illustrated in Example 9.2, which is the dual of the `push` that we presented earlier. When `pop` is executed, the stack is updated as illustrated. Note that We have not written "empty" in memory after the operation because `pop` does not modify the stack memory, although we must assume that it could be modified by a subsequent event.

Putting these together, we can build a skeleton for our `ones` procedure:

```
ones:
       push {r4, lr}     @ save registers
       mov  r4, r0       @ move argument to callee saved register
       ...
       bl   ones         @ recursive call
       ...
       pop  {r4, pc}     @ restore r4 and return
```

Exercise 9.1: Register Save/Restore

Implement the `ones` procedure in assembly and write a C test harness.

Register	Special	Role in Procedure Call Standard
r15	pc	The Program Counter
r14	lr	The Link Register
r13	sp	The Stack Register
r12	ip	The Intra-Procedure-call scratch register
r11		Variable-register 8
r10		Variable-register 7
r9		Platform register (Variable-register 6)
r8		Variable-register 5
r7		Variable-register 4
r6		Variable-register 3
r5		Variable-register 4
r4		Variable-register 1
r3		Argument/scratch register 4
r2		Argument/scratch register 3
r1		Argument/scratch register 2
r0		Argument/scratch register 1

Table 9.1: Core registers and AAPCS usage

ABI Rules for Caller and Callee Saved Registers

The ARM Prouder Call Standard (or ABI) [**?**], defines all of the rules for implementing procedure calls. Among these are the definitions for which registers must be preserved by the caller (caller saved) which must be preserved by the callee (callee saved). Although we restrict our attention primarily to the "low" registers, it is instructive to see the entire register set with their roles:

- The first four registers `r0-r3` are used to pass arguments into a subroutine and return a result from a function. They may also be used to hold intermediate values (between subroutine calls).
- Register `r12` (`ip`) is used by the linker to implement "long" function calls, it may also be used to hold intermediate values.

- Registers `r4-r8` and `r10-r11` may be used by a subroutine to hold its local values. `r9` may have a platform specific use, otherwise it may be used to hold local values. A subroutine (callee) must preserve the contents of registers `r4-r8`, `r9`, and `r10-r11`.

Thus, the *caller saved* registers are `r0-r3` and `r12`. All the other registers below `r13` are *callee saved*. The roles of `r13-r15` are defined.

9.2 Stack Frames

Using the stack to save and restore registers is just one way in which C uses the stack for temporary storage. In this section we present a more general model in which each procedure invocation allocates a *stack frame* on the stack. The stack frame has a standard layout and provides areas for saving registers, local storage, and passing parameters to other procedures. All of these various areas are optional – in the limit, a simple *leaf procedure* might not use the stack at all. [1]

As an example of stack use, we have previously seen that complex expressions can be implemented as a series of simpler expressions with the intermediate values stored in temporary variables. In a compiler, it is the job of the register allocator to determine where these temporary variables are stored – the goal of optimization at this stage is to minimize movement between memory and registers as load/store operations are expensive.

The stack model is illustrated in Figure 9.2. [2] The diagram includes two views of the stack – on the left is a view of the stack prior to a procedure call, and on the right, after a procedure call. In the general model, prior to a call, the caller places parameters on the stack (generally, parameters are passed in a combination of registers and stack memory). The stack pointer indicates the end of the active portion of the stack.

When the call occurs, the first action of the called procedure (the callee) is to save key registers on the stack and then allocate space for local storage – allocation of a parameter area may happen at this point, or prior to a subsequent procedure call. We have previously seen how registers are saved on (restored from) the stack using `push` and `pop` operations. Space for local storage and parameters is allocated (deallocated) by subtracting (adding) an

[1] A leaf procedure is one that calls no other procedures.

[2] This figure, which relatively standard in form, was derived from the Apple document "ARMv6 Function Calling Conventions."

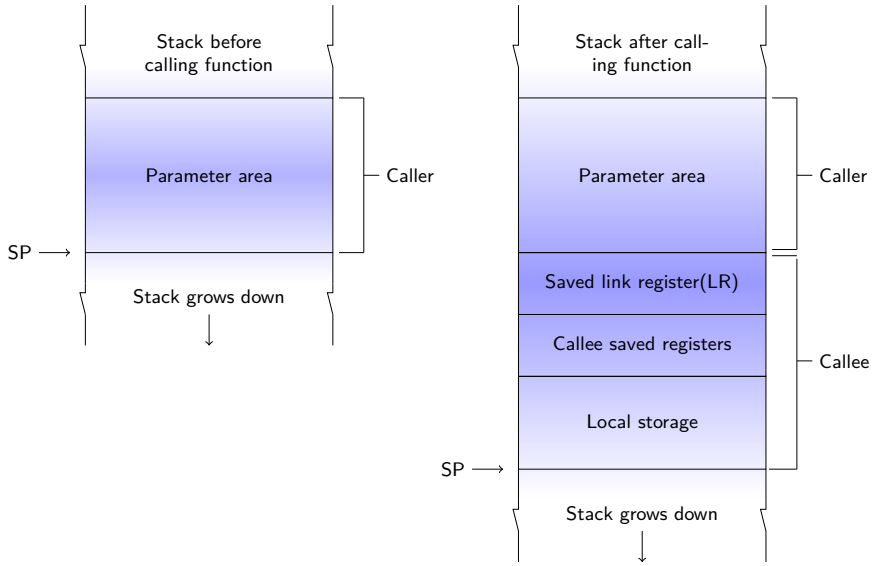


Figure 9.2: Stack Layout

appropriate constant to the stack. The cardinal rule of stack allocation is to never attempt to read information from below (at a lower address) the stack pointer – this area is considered "garbage".

It is common to refer to the code used to set up a stack frame at entry to a procedure as a *prolog*; similarly, the code at exit is referred to as an *epilog*. Consider the example in Example 9.3. The procedure `alloc` accepts one parameter (`i == r0`), allocates a local array, and calls (`foo`). The prolog code pushes `lr` and `r4` (used to preserve `i`), and allocates space for the array `local`. Although, `local` only *requires* 60 bytes, **one requirement of the ARM ABI is that the stack be maintained on 8 byte boundaries.** The epilog deallocates the storage for local and pops the saved `r4` and `lr` (into `pc`). Notice that allocation (deallocation) is performed by subtracting a constant from (adding to) `sp`.

Example 9.3: Stack Frame Example

```
          extern foo(int *);
          int alloc(int i) {
               int local[15];
               return i + foo(local);
           }
```

```
     alloc:
@ prolog
          push {r4, lr} @ save
          sub  sp, 64   @ allocate array
@ body
          movs r4, r0   @ save i
          add r0, sp, 4 @ compute pointer
          bl foo
          adds r0, r4
@ epilog
          add sp, 64    @ deallocate array
          pop {r4, pc}  @ restore
```

9.3 Access Within the Stack

In the preceding section we described how local space may be allocated within the stack frame; however, we did not describe how that space is accessed by a procedure. In Example 9.3, we allocated a 60-byte array and generated a pointer to this array:

```
add r0, sp, #4  @ compute pointer to array
```

This example gives a clue about the general approach – every locally allocated variable is assigned (by the compiler) to a fixed offset from the stack pointer. In the example, we passed this pointer on to another procedure without accessing the memory to which it is a reference.

There are two Cortex-M0 instructions that allow use to read and write words from the stack.

```
ldr rt, [sp, imm8] @ rt = *(sp+(imm8<<2))
str rn, [sp, imm8] @ *(sp+(imm8<<2))
```

`imm8` is an 8-bit unsigned constant which must be multiple of 4 – the assembler accepts values 0,4,...1024 and stores the appropriately shifted value

Example 9.4: Accessing Local Variables

```
extern void foo(int *);
int local(int i) {
       int l = 3;
       foo(&l);
       return i + l;
}
```

```
local:
   push  {r4, lr}     @ save
   sub    sp, 8       @ allocate space
   movs   r4, r0      @ r4 = i
   add    r0, sp, 4   @ r0 = &l
   movs   r3, #3      @ r3 = 3
   str    r3, [sp,4]  @ *(sp + 4) = r3
   bl     foo
   ldr    r3, [sp,4]  @ r3 = *(sp + 4)
   adds   r0, r4, r3  @ retval = i + l
   add    sp, 8       @ deallocate space
   pop    {r4, pc}    @ restore
```

in the instruction word. Because the stack groes downward, all active stack storage is at a positive offset from the stack pointer.

Notice that `ldr` reads from the pointer computed by adding an offset to `sp` and `str` writes from a pointer.

In Example 9.4 we present a simple example that allocates a single local integer variable (`l`). Notice that the assembly code used `r4` to preserve the value of `i`, and `r3` to hold the value written to or read from `l`. As usual, `r0` is used for passing parameters to and returning values from procedures. This example allocates 8 bytes of local storage – 4 of which are filler to ensure that the stack remains aligned on 8-byte boundaries. The actual memory for `l` is at `sp + 4`.

9.4 Parameter Passing

Throughout this text, we have assumed that parameters are passed to a procedure in registers `r0-r3`. This assumption clearly does not work in general; the goal of this section is to describe a model that can handle more than four parameters and parameters that do not fit in a single register (long

long). We will discuss all the C integral types as well as pointers (arrays), but will continue to ignore structures and unions. Pointers cover the most important use of structure parameters (pass by reference) and it is a rare assembly routine that needs to handle the other cases. .

Refer to the ARM Procedure Call Standard for the general case

Although we have mentioned this before, integral types that are smaller than a word are always converted to word-size quantities when passed into a procedure as parameters. Thus `char` and `short` are converted to `int`; similarly, `unsigned char` and `unsigned short` are converted to `unsigned int`. Of course, the processor doesn't distinguish between `int` and `unsigned int`, but the compiler does. For example, choosing logical shift instructions for the unsigned case and arithmetic shift for the signed case.

Earlier in this chapter we introduced a stack frame model that admits a "parameter area" allocated by the caller of a procedure in the callers stack frame. On entry to a procedure, `sp` points to the lowest address of this area. It is convenient to think of registers `r0-r3` as extensions to the parameter areas – this is illustrated in Figure 9.3.

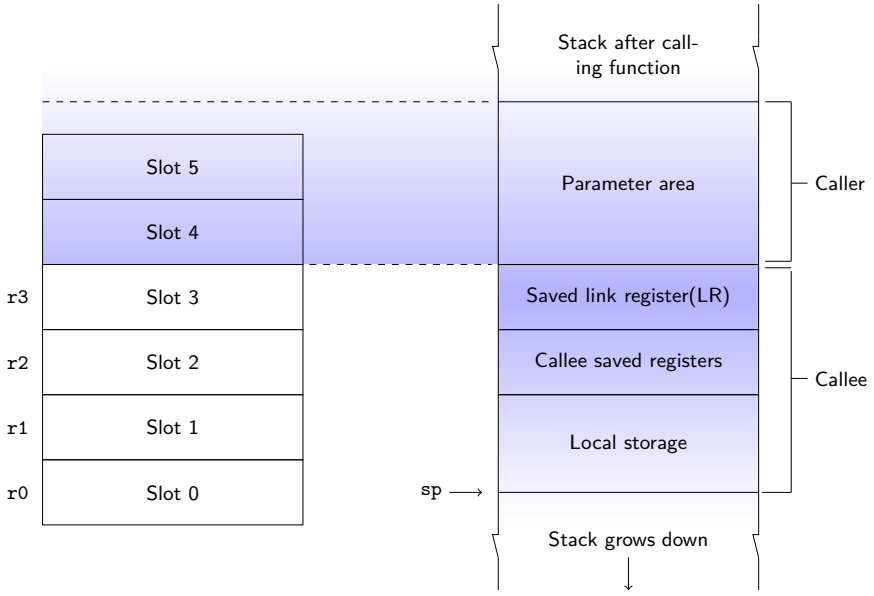


Figure 9.3: Parameter Stack

Suppose that we have a procedure with 6 integer parameters. The first four will be placed in slots 0-3 (`r0-r3`), the remaining two will be placed in

Example 9.5: Accessing Parameters in the Stack

```
extern int foo(int,int,int,int);
int call(int a, int b, int c, int d, int e) {
   return a + e + foo(a,b,c,d);
}
```

```
 call:
    push    {r4, lr}        @ save
    ldr     r4, [sp, #8]    @ r4 = e
    adds    r4, r0          @ r4 += a
    bl      foo
    adds    r0, r4
    pop     {r4, pc}        @ restore
```

slots 4-5. In most cases slot 4 will be at a non-zero offset from `sp`; however, the compiler can compute this offset.

Consider the Example 9.5. Parameters `a-d` are in registers `r0-r3`. Parameter `e` is in slot 4, which is at `sp + 8` rather than `sp` because of the preceding `push` operation. This rather contrived example passes through `a-d` to `foo`.

9.5 Returning Results

– talk about results other than int.

Chapter 10

Exceptions and Interrupts

Chapter 11

Threads

Chapter 12

C Start Code

Chapter 13

Other Instructions

Appendix A

Cortex M0 Instruction Set Summary

The Cortex-M0 instruction set is a subset of the general thumb instruction set. It can be difficult to determine from the various ARM documentation exactly what operands are permissible with which operations (especially for arithmetic operations). If in doubt, try assembling a small test program with the gnu assembler. The following table lists all of the available M0 instructions including several not discussed in this book. `Hi` (`Lo`) refers to registers `r8-r15` (`r0-r7`). Any status flags modified by the instruction are noted.

Operation	Description	Syntax	Flags
Move	8-bit immediate	`movs rd, imm`	N,Z
	Lo to Lo	`movs rd, rm`	N,Z
	Lo to Hi or Hi to Lo	`mov rd, rm`	N,Z
Add	3-bit immediate	`adds rd, rn, imm`	N,Z,C,V
	All registers Lo	`adds rd, rn, rm`	N,Z,C,V
	One register Hi	`add rd, rn, rm`	-
	Any	`add rd, rn`	-
	8-bit immediate	`adds rd, imm`	N,Z,C,V
	immediate to SP	`add sp, imm`	-
	form address from sp	`add rd, sp, imm`	-
	with carry	`addcs rd, rm`	N,Z,C,V
	form address from pc	`adr rd, label`	-
Subtract	3-bit immediate	`subs rd, rn, imm`	N,Z,C,V
	All registers Lo	`subs rd, rn, rm`	N,Z,C,V
	8-bit immediate	`subs rd, imm`	N,Z,C,V

Operation	Description	Syntax	Flags
	immediate from SP	`sub sp, imm`	-
	with carry	`sbcs rd, rm`	N,Z,C,V
	negate	`rsbs rd, rn, 0`	N,Z,C,V
Multiply	32-bit multiply	`muls rd, rm, rd`	N,Z
Compare	Compare	`cmp rn, rm`	N,Z
	Negative	`cmn rn, rm`	N,Z
Logical	AND	`ands rd, rm`	N,Z
	Bit clear	`bics rd, rm`	N,Z
	Exclusive OR	`eors rd, rm`	N,Z
	Move NOT	`mvns rd, rm`	N,Z
	OR	`orrs rd, rm`	N,Z
	AND Test	`tst rd, rm`	N,Z
Shift	Logical shift left	`lsls rd, rm, imm,`	N,Z,C
	Logical shift left	`lsls rd, rm,`	N,Z,C
	Logical shift right	`lsrs rd, rm, imm,`	N,Z,C
	Logical shift right	`lsrs rd, rm,`	N,Z,C
	Arithmetic shift right	`asrs rd, rm, imm,`	N,Z,C
	Arithmetic shift right	`asrs rd, rm,`	N,Z,C
Rotate	Rotate Right	`rors rd, rs,`	N,Z,C
Load	Word, immediate offset	`ldr rd, [rn, imm]`	-
	Word, register offset	`ldr rd, [rn, rm]`	-
	Halfword, immediate offset	`ldrh rd, [rn, imm]`	-
	Halfword, register offset	`ldrh rd, [rn, rm]`	-
	Byte, immediate offset	`ldrb rd, [rn, imm]`	-
	Byte, register offset	`ldrb rd, [rn, rm]`	-
	Signed halfword, register offset	`ldrsh rd, [rn, rm]`	-
	Signed byte, register offset	`ldrsb rd, [rn, rm]`	-
Load	PC-relative	`ldr rd, label`	-
	SP-relative	`ldr rd, [sp, imm]`	-
	Multiple, exclude base	`ldm Rn!, {loreglist}`	-
	Multiple, include base	`ldm Rn, {loreglist}`	-
Store	Word, immediate offset	`str rd, [rn, imm]`	-

Operation	Description	Syntax	Flags
	Word, register offset	`str rd, [rn, rm]`	-
	Halfword, immediate offset	`strh rd, [rn, imm]`	-
	Halfword, register offset	`strh rd, [rn, rm]`	-
	Byte, immediate offset	`strb rd, [rn, imm]`	-
	Byte, register offset	`strb rd, [rn, rm]`	-
Store	SP-relative	`str rd, [sp, imm]`	-
	Multiple	`stm Rn!, {loreglist}`	-
Push	Push	`push {loreglist}`	-
	Push with lr	`push {loreglist,lr}`	-
Pop	Pop	`pop {loreglist}`	-
	Pop with pc	`pop {loreglist,pc}`	-
Branch	Conditional	`b<cc> label`	-
	Unconditional	`b label`	-
	With link	`bl label`	-
	With exchange	`bx rm`	-
	With link and exchange	`blx rm`	-
Extend	Signed halfword to word	`sxth rd, rm`	-
	Signed byte to word	`sxtb rd, rm`	-
	Unsigned halfword to word	`uxth rd, rm`	-
	Unsigned byte to word	`uxtb rd, rm`	-
Reverse	Bytes in word	`rev rd, rm`	-
	Bytes in both halfwords	`rev16 rd, rm`	-
	Signed bottom halfword	`revsh rd, rm`	-
State Change	Supervisor call	`svc imm`	-
	Disable interrupts	`cpsid i`	-
	Enable interrupts	`cpsie i`	-
	Read special register	`mrs rd, specreg`	-
	Write special register	`msr specreg, rn`	-
	Breakpoint	`bkpt imm`	-
Hint	Send event	`sev`	-
	Wait for event	`wfe`	-

Operation	Description	Syntax	Flags
	Wait for interrupt	`wfi`	-
	Yield	`yield`	-
	No operation	`nop`	-
Barriers	Instruction synchronization	`isb`	-
	Data memory	`dmb`	-
	Data synchronization	`dsb`	-

Appendix B

The gnu-arm Toolchain

B.1 Introduction

Major components – cpp, gcc, gas, ld, binutils, make, qemu, gdb.

B.2 Installing

B.3 Tool Flow and Intermediate Files

B.4 An Extended Example

Appendix C

Test Framework

C.1 A Test Framework

Programming assembly language is sufficiently different from programming conventional langauges that it is important to test your understanding by executing small programs within a debugger so that you can examine the processor state (memory and registers) throughout the execution process. In a conventional programming language, most "state" (both data and control) is implied while in assembly langauge all of the state is exposed to the programmer. For example, consider a simple variable – in a language such as C or Java you use the textual name to refer to the variable. In assembly language, we always reference a variable by its address.

The methodolgy that we use throughout this book is to test assembly programs within an emulator (qemu) under the control of a debugger (gdb). Furthermore, we wrap most assembly code in a C test harness that allows us to easily create test cases and print meaningful execution information. This test harness is compiled along with our assembly code and executed within the emulator – the assembly code is treated as a procedure by the C test harness.

Consider the short program illustrated in Figure C.1. The program consist of a few assembly language directives – `.syntax unified` which tells the assember we are using the newer "unified" syntax for assembly [1], `.text`, which tells the assembler that the following code belongs in the text segment, `.thumb`, which tells the assembler to generate 16-bit code, `.align 2`, which enforces 16-bit alignmnent, and `.global main`, which declares a global variable. These are followed by a label, three assembly instructions, and a final

[1] It is important that you use this directive for any of the examples in this book !

directive indicating the end of the file. The first instruction, `nop`, does nothing but introduce an instruction delay so that when stepping through the code, the debugger has a convenient "landing" spot. The third instruction, returns from the main procedure to the initialization code linked by default. The the single add instruction represents our test code.

```
  .syntax unified
  .text
  .thumb          @ thumb instruction set
  .align 2        @ force half-word alignment
  .global main    @ symbol declaration

main:
  nop             @ landing spot for gdb
  adds r0, 1      @ example code
  mov  pc,lr      @  return
  .end
```

Figure C.1: Simple Assembly Test

This code template can be compiled as:

```
arm-none-eabi-gcc -g -mcpu=cortex-m0 -mthumb  -o template.elf \
                  test.s -specs=rdimon.specs -lc -lrdimon
```

The resulting object `template.elf` can be executed and debugged with the following commands executed in separate terminal windows:

```
qemu-system-arm -cpu cortex-m3  -semihosting  -S \
                -gdb tcp::51234 -kernel template.elf


arm-none-eabi-gdb -ex ``target remote localhost:51234'' \
                  -ex ``load'' template.elf
```

This will allow you to run and step through the assembly program. A somewhat more interesting use of these tools is to build a test harness in C which calls an assembly language procedure with test data and prints the results (the assembly routine differs from the previous one only in the global name used – test vs. main).

```
#include <stdio.h>
#include <limits.h>

extern int test(int);
int data[] = {INT_MIN,  -1, 0, 1, INT_MAX};
void main() {
  int i;
  for (i = 0; i < sizeof(data)/sizeof(int); i++)
```

```
    printf("Input %d output %d\n", data[i], test(data[i]));
}
```

```
  .syntax unified
  .text
  .thumb          @ thumb instruction set
  .align 2        @ force half-word alignment
  .global test    @ symbol declaration

test:
  nop             @ landing spot for gdb
  adds r0, 1      @ example code
  mov  pc,lr      @  return
  .end
```

The resulting binary can be executed as:

```
qemu-system-arm -cpu cortex-m3  -semihosting -kernel testharness.elf
```

With the following output

```
Input -2147483648 output -2147483647
Input -1 output 0
Input 0 output 1
Input 1 output 2
Input 2147483647 output -2147483648
```

The toolchain libraries provide a full implementation of the standard C libraries and the qemu/arm "semihosting" feature provides access to the host operating system for standard file and other system calls.